Mapping Objective-C API to Java

Arno Puder and Spoorthi D'Silva

San Francisco State University
Department of Computer Science
1600 Holloway Avenue
San Francisco, CA 94132
{arno,spoorthi}@sfsu.edu

Abstract. Apple champions the use of Objective-C for iOS development and has prohibited the use of virtual machines in the past. In previous work we have shown how Java can be used as an alternative programming language for iOS applications. We described a byte code-level cross-compiler that translates Java-based applications to portable C code to circumvent this legal restriction. A major challenge not addressed in our previous work pose the significantly growing size of the iOS API, since every Objective-C based API needs to be exposed in Java. So far we required the necessary Java skeletons to be written by hand. Since its introduction in 2007, the iOS API has nearly doubled in size. Considering the size of the iOS API this approach does not scale. In this paper we describe an API mapping tool that can generate the required Java skeletons by parsing Objective-C header files. Emphasis is placed on mapping Objective-C's dynamically typed API to strongly typed API in Java.

Keywords: API Mapping, Objective-C, Java, iOS.

1 Introduction

With the introduction of the iPhone in 2007 the smartphone market has witnessed significant growth, slowly cutting into the traditional cell phone business. Apple has created an immensely successful ecosystem around iOS with the AppStore. Apple offers an Objective-C based development environment for iOS apps. The tremendous growth of the iOS platform is reflected by the size of its API (Application Programming Interface). Table 1 gives a quantitative comparison on the number of classes and methods between different versions of iOS. It should be noted that iOS still exposes a significant amount of C API via functions and structs besides the Objective-C based API. As can be seen, the iOS API has nearly doubled in size since its first public release.

Apple has long resisted the use of programming languages other than Objective-C. Amongst others, Apple forbade the use of virtual machines on iOS. In a previous project, we designed and implemented a byte-code level cross-compiler to translate Java classes to portable C code, thus circumventing Apple's restrictions regarding virtual machines [12]. Cross-compilation is necessary but

D. Uhler, K. Mehta, and J.L. Wong (Eds.): MobiCase 2012, LNICST 110, pp. 21–43, 2013.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

not sufficient for developing Java-based iOS applications. An application also depends on the rich iOS API that must be exposed in Java.

In the past, we have manually designed and written Java skeletons to provide a Java API for iOS. Considering the substantial size of the iOS API, the process of manually writing these skeletons does not scale. As a continuation of our previous work, we describe a API mapping tool in this paper that parses Objective-C header files to generate Java skeletons. Doing so requires to solve two major issues: firstly, mapping API based on a dynamically-typed language (Objective-C) to a statically-typed API (Java). Secondly, creating "glue code" that bridges between the different object models of the two languages at runtime.

Table 1. iOS API statistics

	iOS 3.2	iOS 4.2	iOS 5.0
Classes	282	423	503
Protocols	63	93	112
Instance Methods	4357	5756	7021
Static Methods	470	637	759
Functions	2041	2484	2714
Structs	236	289	426

This paper is organized as follows: Section 2 provides a brief overview of Objective-C and iOS. Section 3 discusses related work. In Section 4 we give a detailed overview of our API mapping tool. Section 5 provides a side-by-side comparison of a simple application written in Objective-C and the same application written in Java using our generated API. Finally, in Section 6 we give a brief conclusion and an outlook to future work.

2 Background

In this section we provide a very high-level overview of Objective-C and iOS. It is not meant as an exhaustive introduction to either topic which is out of scope for this paper. The purpose of this section is to give sufficient background information on both Objective-C and iOS to motivate the challenges mapping Objective-C API to Java. The examples given in the following highlight the corner cases that must be dealt with during the API mapping. A subsequent section will describe our solutions to the various challenges.

2.1 Objective-C

Objective-C was initially designed by Brad Cox in the early eighties. It is inspired by Smalltalk basing its object model on dynamic typing. NeXT Software licensed the Objective-C language in 1988 and developed its operating system called NEXTSTEP. In the mid-nineties, Apple acquired NeXT Software and the

language became the language of choice for application development. With the introduction of iOS in 2007 the language has seen a renaissance and has become the main development language for iOS app development. Apple offers its own integrated development environment based on Objective-C through Xcode.

Similar to C++, Objective-C is a strict superset of the C programming language adding object oriented features [7]. Any legal C program is also a legal Objective-C program. All C libraries are available in Objective-C and the entry point of an Objective-C program is defined via the usual `main()` function. The following code excerpt shows how to define a class in Objective-C:

Objective-C class declaration and definition

```
1 @interface Fraction: NSObject {
2     int numerator;
3     int denominator;
4 }
5 -(void) setNumerator:(int)n;
6 -(void) setDenominator:(int)d;
7 -(int) numerator;
8 -(int) denominator;
9 @end
10
11 @implementation Fraction
12 -(void) setNumerator:(int)n { numerator = n; }
13 -(void) setDenominator:(int)d { denominator = d; }
14 -(int) denominator { return denominator; }
15 -(int) numerator { return numerator; }
16 @end
17
18 // Using class Fraction
19 Fraction* frac = [[Fraction alloc] init];
20 [frac setNumerator:3];
21 [frac setDenominator:5];
22 printf("Numerator: %d", [frac numerator]);
```

Just like C++, Objective-C distinguishes between a *declaration* (denoted by the keyword `@interface`) and a *definition* (denoted by the keyword `@implementation`) of a class. Class `NSObject` serves as a base class, similar to `java.lang.Object`. The prefix "NS" is common for many Objective-C class names and is a relict of the origins of the class library developed by NeXT Software. Fields are declared within curly brackets followed by the methods of the class. Instance methods are prefixed with a "-" while class methods are prefixed with a "+" followed by the return type of the method. Lines 19–22 show the use of an Objective-C class. The square brackets denote a method invocation where the first argument is the target followed by the message being sent to that target. Creating a new instance requires to send the class object the `alloc` message (which only allocates sufficient memory for the instance) followed by the `init` message that serves as a constructor. Objective-C supports the notion of *named arguments*. Unlike other programming languages such as C, C++, or Java, in

Objective-C every argument is named as shown in the following addition to class `Fraction`:

Named Arguments

```
1 // ...
2 -(void) setNumerator:(int)n andDenominator:(int)d
3  { numerator = n; denominator = d; }
4
5 // ...
6 [frac setNumerator:3 andDenominator:5];
```

Named arguments support the notion of *fluent interfaces* that make it easier for a programmer to associate semantics with a method [4]. The concatenation of the argument names `setNumerator:andDenominator:` is referred to a *selector* in Objective-C. Method dispatch is solely based on the selector. Since formal parameter types play no role in method dispatch, Objective-C does not support overloading. Mapping a selector to a Java method name would therefore be valid mapping. However, selectors in iOS tend to be lengthy, such as `tableView:willDisplayCell:forRowAtIndexPath:` from `UITableViewDelegate`. Exploiting Java's capability for method overloading leads to more poignant method names as shown in a subsequent section.

Although Objective-C is dynamically typed, it is possible to define interfaces to allow for better checks at compile time. Interfaces are defined via so-called *protocols*. While Objective-C only supports single inheritance for classes, it supports multiple inheritance for protocols. The following code excerpt shows the declaration of the protocol `Printing` that is implemented by class `Fraction`:

Objective-C protocols

```
1 @protocol Printing
2 -(void) print;
3 @end
4
5 @interface Fraction: NSObject <Printing>
6 // ...
7 @end
8
9 @implementation Fraction
10 -(void) print
11  { printf("%d/%d", numerator, denominator); }
12 @end
13
14 // Call print method
15 id<Printing> obj = ...;
16 [obj print];
```

The protocols that a class implements are listed between $<$ and $>$ in the class declaration. Protocols are not first class types, i.e., they cannot be used as

type names. The declaration in line 15 above tells the Objective-C compiler that variable `obj` is a reference to an object supporting the `Printing` protocol. It is important to note that true to its dynamic nature, a class may not implement all methods declared in a protocol. Invoking an unimplemented optional method leads to a runtime error. iOS is making liberal use of optional methods. Since Java does not support optional methods in interfaces, this will be a challenge for the API mapping.

Objective-C supports the notion of class mix-ins through *categories* [13]. A category can add methods to an existing class. The name of the category is mentioned between round brackets:

Objective-C categories

```
1 @implementation Fraction (FractionCategory)
2 -(void) multiplyWith:(Fraction*)f
3 {
4     numerator *= f->numerator;
5     denominator *= f->denominator;
6 }
7 @end
```

Although categories cannot add additional fields, they can add one or more methods to an existing class, even to classes for which the source code is not available. In the example above, instances of class `Fraction` can respond to the message `multiplyWith:`. iOS makes use of categories which will need to be mapped to Java.

2.2 iOS

Since its introduction in 2007, the iOS API is largely based on Objective-C. Common to many GUI frameworks, iOS supports the Model/View/ Controller paradigm through various classes [8]. Common UI elements are defined by classes such as `UILabel` (labels), `UIButton` (buttons), or `UITextView` (input fields). Complex widgets such as `UIAlertView`, `UITableView` or `UIDatePicker` have become the trademark look-and-feel for iOS-based apps. The following excerpt from the iOS SDK shows the API to create a new `UIButton` via the static method `buttonWithType:`:

Objective-C: UIButton

```
1 typedef enum {
2     UIButtonTypeCustom = 0,
3     UIButtonTypeRoundedRect,
4     //...
5 } UIButtonType;
6
7 @interface UIButton : UIControl
8 +(id) buttonWithType:(UIButtonType)buttonType
9 //...
10 @end
```

From an API mapping perspective there are two points worth mentioning in this example. For one, even though method `buttonWithType:` returns a `UIButton` instance, the return type is declared as the generic `id` type. For a strongly typed language such as Java it would be more appropriate to map the return type to `UIButton`. Furthermore, the formal type of parameter `buttonType` is based on an enum. Even though the compiler treats the enum as an `int`, the Java API should offer identifiers for constants such as `UIButtonTypeRoundedRect`.

iOS makes heavy use of Objective-C protocols, especially to define the interface of delegates that serve as callbacks. Much of the behavior of UI widgets is driven by delegates that define different aspects of a widget. E.g., protocol `UITableViewDelegate` features 18 callback methods that control the look-and-feel of a `UITableView`. All but one of these callbacks are optional and need not be implemented. Mapping an Objective-C protocol to a Java interface would therefore incur an inconvenience to a Java developer as all 18 methods would need to be implemented.

iOS offers several classes to support various data structures. Class `NSArray` manages an array while `NSDictionary` maintains a set of key-value pairs. These classes are used by iOS GUI classes. E.g., the `UIView` class that represents the base class of the view hierarchy has a method `getSubviews` that returns a `NSArray` instance with a list of all child views. When mapping this method to a Java API the question arises if the Java counterpart should also offer Java versions of classes `NSArray` and `NSDictionary` or use the more familiar Java interfaces `java.util.List` and `java.util.Map`.

As already mentioned in the introduction, a significant portion of the iOS API is not defined through Objective-C classes but C functions. In particular the low-level graphic drawing functions of the Quartz framework are exclusively defined via C functions. As an example, consider the following iOS API part of the core graphics framework that defines a rectangular region on the screen:

C: CGRect

```
1 struct CGRect {
2     CGPoint origin;
3     CGSize  size;
4 };
5 typedef struct CGRect CGRect;
6
7 CGRect CGRectIntersection (
8     CGRect r1,
9     CGRect r2
10 );
```

A `struct` in Objective-C retains its original semantics of the C programming language and can be viewed as a value type. A set of C functions perform various operations on `CGRect` instances such as computing the intersection of two rectangles. As Java does not support functions or value types, special mapping rules must be devised to yield a natural Java API. The iOS API also makes use of features typically found in C programs. The following code excerpt shows

the use of a pointer-to-a-pointer argument to mimic call-by-reference which also needs to be handled specially in Java:

Call-by-reference
```
1 // Objective-C
2 @interface AVAudioSession : NSObject
3 -(BOOL) setActive:(BOOL)beActive
4             error:(NSError**)outError;
5 //...
6 @end
```

Objective-C categories are used in iOS to define so-called *additions* that add features to existing classes. E.g., iOS's UIKit adds several methods to class `NSString` that are specific to UI programming. As shown below, the `UIKitAddition` category adds a method to `NSString` to determine the width and height in pixels of a string given a particular font:

UIKit addition to NSString
```
1 @interface NSString (UIKitAddition)
2 -(CGSize) sizeWithFont:(UIFont*)font;
3 //...
4 @end
```

Finally we want to mention the memory management model of iOS applications that has no impact on the generated Java API, but has significant consequences of the underlying JNI implementation. Apple has opted not to include a garbage collector in iOS. Instead, memory management is handled via reference counting [6]. Base class `NSObject` offers methods `retain` and `release` to increase and decrease the reference count of an object respectively:

Reference counting
```
1 Fraction* frac = [[Fraction alloc] init];
2 printf("Retain count: %d", [frac retainCount]);
3 [frac retain];  // 2
4 [frac release]; // 1
5 [frac release]; // 0. Object will be deleted
```

Proper memory management is one of the most difficult aspects of iOS programming. While iOS 5 has introduced ARC (Automatic Reference Counting) where the compiler automatically inserts `retain`/`release` methods, this feature is not supported by the core framework and it will also not work for multi-threaded applications. Making Java's garbage collector work in conjunction with reference counting is a major challenge of the cross-compiled Java application.

In this section we have highlighted various features of the Objective-C programming language and the iOS API that have impact on the design of a corresponding Java API. After discussing some related work, we will describe our

mapping tool that can generate a Java API taking C/Objective-C features such as structs, categories or named parameters into account.

3 Related Work

API mapping from one language to another is not new. There are several ongoing efforts in this area. In this section we will discuss some prominent examples, some of which are not in the mobile space. One example is SWIG (Simplified Wrapper and Interface Generator) that is used to bind the code written in C/C++ to a variety of languages like Tcl, Perl, Java, C#, etc [3,1]. SWIG does this by automatically generating the code required for the binding using a simple interface file as input. It tries to provide a complete mapping for the APIs exposed in this interface file that needs to be provided by a developer. Since SWIG does not have a memory manager of its own, special care needs to be taken in SWIG to avoid releasing the associated C++ objects that still might be used. SWIG provides directives to handle special cases but does not make use of any heuristics for mapping the API.

Similar to SWIG, CORBA (Common Object Request Broker Architecture) helps with mapping API of remote objects in a distributed environment [11]. CORBA makes use of a specialized language called IDL (Interface Definition Language) that can be used to define the interface of remote objects. IDL is not tied to a specific programming language and the CORBA specification defines mapping rules from IDL to various high-level languages. An IDL compiler creates stubs and skeletons from a specific IDL file that are used to carry out the client server communication. Stubs are generated in the same language as the client while the skeletons are generated in the same language as the server. Client and server do not need to be implemented in the same language.

While the tools discussed so far do not directly relate to mobile applications, there are tools such as MonoTouch that allow developers to create iOS applications using C# and .NET [14]. MonoTouch uses a so-called API Definition file that consists of C# interfaces with special annotations and attributes for the binding purpose. These special annotations handle the exceptional cases for the API mapping such as static methods, binding rules for properties and so on. MonoTouch has its own garbage collector and also provides users with options to release resources explicitly before the garbage collectors collects them.

While MonoTouch concentrates on offering a binding between the C# and Objective-C, there are tools that try to provide cross-platform support. One such tool is PhoneGap that supports JavaScript API that acts as the bridge to the native platform [2]. PhoneGap provides functionality to develop more than just a web app in the sense that the JavaScript APIs also provide access to the phone's hardware such as accelerometer, camera, etc.

PhoneGap provides a generic interface across multiple platforms using HTML, JavaScript and CSS. By default, the API support is only for certain hardware functions and and does not cover a 1:1 mapping of the various mobile platform APIs. One thing to note is that PhoneGap does support a wide range of

mobile platforms like iOS, Android, Blackberry to name a few. In order to perform complex tasks, a developer can write custom plugins by writing custom JavaScript APIs and custom native components so that JavaScript invocations can be delegated to the native code.

While PhoneGap and MonoTouch provide native API access in a different programming language, JamVM, a light-weight implementation of the Java Virtual Machine enables developers to run Java programs on iOS [10]. JamVM for iOS uses reflection in Objective-C to extract API information from the binary iOS libraries. Based on that information, JamVM for iOS generates JNI code that a Java application needs to make calls to iOS. Since JamVM for iOS effectively parses API from a binary library, it is impossible to extract symbolic information such as type identifiers resulting in unnatural looking Java API. Furthermore, JamVM for iOS is not capable of generating Java API from C functions.

Unlike some of the tools mentioned above that only map a subset of the native APIs, our tool tries to provide complete mapping for all the APIs that are available in iOS. Our tool automates the process of generating the API mappings. It not only handles exceptions by providing special hints but also tries to create more natural looking APIs to support Java-ism by using special heuristics. The application developer need not take special care for releasing the resources while using our tool since our API mapping tool provides its own memory management mechanism.

Table 2 summarizes the features of the various tools discussed above and compares them to our API mapping tool.

Table 2. Comparison of related work

	SWIG	**CORBA IDL**	**MonoTouch**	**PhoneGap**	**JamVM for iOS**	**Our API mapping tool**
Input Artifact	Header	IDL	Header	n.a	Binary Library	Header
API mapping mechanism	Automated	Automated	Automated	Manual	Automated	Automated
Language Association	1:n	m:n	1:1	n:1	1:1	1:1
Memory Management	Manual	Manual	Automatic	Automatic	Automatic	Automatic
Handling Anomalies	Directives	n.a	Annotations	n.a	Not Present	Advisor
Scope	Complete	Complete	Complete	Subset	Subset	Complete
Heuristics for natural looking API	✗	✗	✗	n.a	✗	✔

4 API Mapping

This section introduces our API mapping tool. Section 4.1 will first discuss the design goals and the architecture of our tool. Section 4.2 describes the Java API that our tool generates. The Java API needs to be complemented by matching JNI code that acts as a bridge between the Java application and iOS.

The generation of the JNI code is described in Section 4.3. In Section 4.4 we give a brief overview of the external advise needed by our tool.

4.1 Overview

Before giving an in-depth overview of our API mapping tool, we first need to establish the goals of its design. Ultimately the purpose of the tool is to avoid having to manually implement the Java API whenever Apple releases a new version of iOS. At the same time, Java and Objective-C are quite different languages in the way they implement their respective object model. When deriving a Java API from its Objective-C original, one should take those differences into account. Specifically, our mapping tool strives to fulfill the following design goals:

1. Automate API mapping.
2. Map all relevant iOS API.
3. Keep naming conventions.
4. Support Java-isms.

First and foremost the API mapping should be automated. Given the fact that iOS evolves rapidly from version to version manually mapping new Objective-C API to Java does not scale. Our second goal is to map all relevant iOS API to increase coverage. This implies that we not only map Objective-C API but also the significant number of C functions part of the iOS API. Our third goal is to make the transition from Objective-C to Java as seamless as possible. By keeping naming conventions it will be easier to transition to Java for iOS development as the original documentation from Apple can serve as a reference even for the Java API.

The most important goal is to support conventions that Java developers have become accustomed to. For one, the API should make use of strongly typed interfaces where appropriate. Due to the nature of Objective-C's dynamic typing, the API is often less specific compared to an equivalent Java API. Furthermore, we try to make use of familiar J2SE API instead of iOS specific data structures. Last but not least, Java developers expect the presence of a garbage collector that will need to interface with the reference counting mechanism used by iOS.

Figure 1 gives an overview of the architecture of our API mapping tool. Common to any compiler, the mapping tool can be divided into a frontend and a backend. The frontend reads and parses the original iOS SDK header files to scan it for relevant API. As mentioned earlier, not all required information can be derived from those header files (such as the specific types of arguments). In these cases, external *advise* needs to be provided to the mapping tool. Based on the header files and the advise, the backend generates two kinds of artifacts: the Java API that can be used by an iOS app developer and the JNI code that acts as a bridge between the Java application and iOS. All aspects of the API mapping tool are discussed in detail in the following sections.

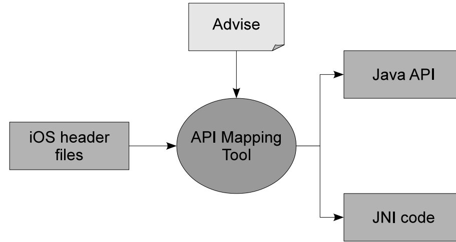

Fig. 1. API Mapping Tool

4.2 Generating Java API

In the following we focus on how a Java API can be derived from the original C/Objective-C API. As C is the foundation of Objective-C, we first discuss how to map certain features of C to Java followed by a description of the mapping of Objective-C API.

Mapping of C API. Primitive types are mapped to their natural counterparts. It is common in C to typedef names for commonly used primitive types since their size is not defined by the C standard (e.g., `sizeof(int)` depends on the underlying architecture). iOS has typedefs for various primitive types such as `BOOL`, `int8_t`, `int16_t`, and `int32_t` that are mapped to their Java counterparts `boolean`, `byte`, `short`, and `int`. Signed and unsigned integer values have to be treated the same as Java does not offer support for unsigned values. Defining a platform-independent type such as `int32_t` is usually done by `#ifdef`s that check for different architectures. In order to avoid parsing preprocessor directives, we provide this knowledge via external advise.

Many iOS data structures are defined via C structs such as `CGRect` introduced in Section 2.2. A `CGRect` defines a 2-dimensional region in the iOS core graphics framework. Its members are based on two other structs, `CGPoint` and `CGSize`, for the position and the size of the region. Since Java does not offer support for value types, structs are mapped to Java classes, making the members of the struct public fields of these classes.

The iOS API uses very regular and dependable naming conventions. In Section 2.2 we mentioned function `CGRectIntersection` that computes the intersection of two `CGRect` instances. The name of all C functions that operate on a particular struct are always prefixed with the name of that struct in iOS. The first argument is an instance of that struct. We take advantage of these regular naming conventions by making these C functions the methods in the generated Java class. The first parameter becomes the implicit `this`-parameter. The following code excerpt shows the generated Java version of `CGRect`:

Java version of CGRect
```
1 public class CGRect {
2     public CGPoint origin;
3     public CGSize  size;
4 
5     native public CGRect intersection(CGRect r2);
6     //...
7 }
```

We take advantage of similar naming conventions when mapping enums. The common prefix of enum members is removed since Java enums define a scope in contrast to C enums. The following code shows the mapping of `UIButtonType` introduced in Section 2.2:

Mapping of an enum
```
1 // Objective-C
2 typedef enum {
3     UIButtonTypeCustom = 0,
4     UIButtonTypeRoundedRect,
5     //...
6 } UIButtonType;
7 
8 // Java
9 public enum UIButtonType {
10     Custom(0),
11     RoundedRect,
12     //...
13 };
```

It should be noted that the rules to take advantage of naming conventions are hard-coded heuristics and not part of the advise used by our tool. If an API does not make use of those naming conventions, functions are mapped to static methods of a global class.

Mapping of Objective-C API. By default, an Objective-C class is mapped to a Java class of the same name. A selector in Objective-C is mapped to a Java method. While it would be possible to use the selector as the name of a Java method, it would result in unnatural names. Since Java does not support named arguments, using the selector as the name for a method could be confusing. We use various heuristics to generate more Java-like method names taking advantage of method overloading in Java. By default, the first component of a selector is used as the name of the method. As an example, consider the following methods of class `NSString`:

Mapping of NSString
```
1 // Objective-C
2 - (int)compare:(NSString *)string;
3 - (int)compare:(NSString *)string
```

```
4        options:(NSStringCompareOptions)mask;
5 - (int)compare:(NSString *)string
6        options:(NSStringCompareOptions)mask
7          range:(NSRange)compareRange;
8 - (int)compare:(NSString *)string
9        options:(NSStringCompareOptions)mask
10         range:(NSRange)compareRange
11        locale:(id)locale;
12
13 // Generated Java methods
14 public int compare(NSString string);
15 public int compare(NSString string, int mask);
16 public int compare(NSString string, int mask,
17                    NSRange compareRange);
18 public int compare(String string, int mask,
19                    NSRange compareRange,
20                    Object locale);
```

Class `NSString` features four different methods to compare two strings. Instead of using the selector as a method name only the first component of the selector is used. For this reason selector `compare:options:` is mapped to method name `compare` in Java. This is possible because the formal parameter types are different for all four selectors and one can rely on Java's method overloading to distinguish among them. If the formal parameter types are identical for different selectors, our heuristic will append the second component of the selector to the generated method name until the Java methods are distinguishable again.

Parameter types are mapped to their counterpart. The Objective-C header parser resolves typedefs to determine the underlying type. Since Java does not support typedefs, the original type is used when generating the Java API. This can be seen in the example above where `NSStringCompareOptions` is a typedef for an `int`.

Various other heuristics determine the mapping for formal parameter types. E.g., the generic Objective-C type `id` is mapped to `java.lang.Object`. For parameters whose type is a pointer-to-a-pointer we assume that it is an output argument. In this case a so-called *holder class* is generated that acts as a wrapper for the actual output value. We make use of Java generics to create a type-safe parameter type. As an example, `NSError**` in the Objective-C API would be mapped to `Reference<NSError>` on the Java side.

As expressed in the design goals for the API mapping, we strive towards natural looking Java APIs. For this reason Objective-C data structures are mapped to their J2SE counterpart. E.g., `NSArray` will be mapped to `java.util.List` and `NSDictionary` will be mapped to `java.util.Map`. There are three main reasons for doing so:

Usability: Java programmers know how to deal with Java data structures. They know the API and are good at using it. Using the original Objective-C

requires to learn new API. Furthermore, making use of J2SE API has the benefit of being able to use Java-isms such as for-each loops.

Performance: Cross-compiling a Java implementation for `java.util.List` to C code is more efficient than having a wrapper for `NSArray` where each method invocation on the Java side has to be forwarded to an appropriate method of `NSArray`. The one extra level of indirection incurs avoidable overhead.

Scope: The question arises where to draw the line. E.g., using a Java version of `NSString` instead of `java.lang.String` would result in unnatural Java programs, especially since Java string literals are of type `java.lang.String`.

For those reasons we have opted to map Objective-C classes to their natural J2SE counterpart. Java versions of classes such as `NSArray` and `NSString` are available, however, if used as formal argument types in method signatures they are mapped to the equivalent J2SE type. Mapping of generic parameter types such as `NSArray` and `NSDictionary` leads to another challenge. Due to Objective-C's support for dynamic typing those data structures can hold objects of arbitrary type. This would be equivalent to `List<Object>` and `Map<Object, Object>` in Java. However, in many cases the type of the objects held in a container is restricted. E.g., the `getSubviews` method of class `UIView` returns a list of children of a particular `UIView` instance. As per iOS documentation each child must be an instance of a subclass of `UIView`. Objective-C is not capable of expressing this restriction in a method's signature. However, in Java it would be possible to restrict the return type of `getSubviews` by making use of generics to `List<UIView>`. We have opted to make use of Java generics and support the mapping of restricted container types. Since this information is not discernable from the Objective-C header files of iOS, it must be provided externally as an advise to the API mapping tool.

Categories were introduced as a mechanism in Objective-C to add methods to existing classes and as was shown in Section 2.2, iOS makes use of this in the UIKit Additions. Since Java does not support mix-in classes, the way we handle additions is by inlining the methods to the class to which they are added via a category. Another challenge is the mapping of Objective-C protocols. By default, an Objective-C protocol is mapped to a Java interface. This requires an app developer to implement all methods of the interface. Since many methods are often optional in iOS, we also emit an *adapter class* in Java that implements the interface [5]. Optional methods are given a default implementation in the adapter class and only mandatory methods are left abstract.

4.3 Generating Java JNI

The previous section discussed the mapping of Objective-C API to Java. The result is a natural looking Java API for iOS that can be used by an app developer. Methods of the Java skeleton classes are labeled as `native` and running a Java-based iOS app requires JNI code that acts as a bridge between the Java application and iOS [9]. Our mapping tool also generates the necessary JNI code automatically. The details are discussed in the following.

Wrapping. When a Java application uses a class from the iOS library, it inadvertently needs to use a native Objective-C object. As an example, consider class `UIAlertView`. In iOS, this class is used to pop open an alert view as a modal dialog, often presenting the user with several options. Instantiating the Java class `UIAlertView` must lead to an instantiation of the Objective-C class `UIAlertView`. Subsequent method invocations on the Java object need to be forwarded to its Objective-C counterpart. The following code shows the effect of executing Java code in relationship to what needs to happen on the native layer:

Effects of Java method invocations
```
1 // Java (Configuration of UIAlertView not shown here)
2 UIAlertView alert = new UIAlertView();
3 alert.show();
4
5 // Objective-C
6 UIAlertView* alert = [[UIAlertView alloc] init];
7 [alert show];
```

The Java object can be seen as a wrapper or proxy for the underlying native Objective-C object. Method invocations on the Java object are forwarded to the wrapped Objective-C object. This happens transparently to the app developer. Figure 2 illustrates the notion of a wrapper. White circles represent pure Java objects. Those objects are part of the application and have no relationship with iOS. Objective-C objects are represented by black dots. Some Objective-C objects exist only inside iOS and are not visible to the application. Other Objective-C objects such as the `UIAlertView` instance are exposed to the Java application via a wrapper.

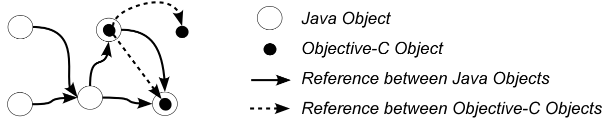

Fig. 2. Wrapping native Objective-C objects

It should be obvious that there needs to be a tight association between a wrapper object and the wrapped native Objective-C object. This association is bijective and it must be possible to access the native object from the wrapper and vice versa. We make use of JNI to forward an invocation to a Java method to the underlying Objective-C object. All methods in the generated Java API are declared as native to allow the injection of Objective-C code. The following code excerpt shows the native implementation of the two methods of `UIAlertView` used in the example earlier:

JNI implementation
```
1 void UIAlertView___INIT___(JAVA_OBJECT me)
2 {
```

```
3     UIAlertView* obj = [[UIAlertView alloc] init];
4     ASSOCIATE(me, obj);
5 }
6
7 void UIAlertView_show__(JAVA_OBJECT me)
8 {
9     UIAlertView* thiz = GET_ASSOCIATED_NATIVE(me);
10    [thiz show];
11 }
```

The JNI code shown here deviates from the official JNI specification in order to simplify the example. Note that JNI requires C functions for native Java methods, however, it is possible to make use of Objective-C for the implementation of those functions. Instantiating class `UIAlertView` yields in the invocation of `UIAlertView___INIT___` via JNI where the corresponding Objective-C version of `UIAlertView` is created via the usual `alloc/init` messages. Function `ASSOCIATE` is part of our runtime library and its purpose is to create the aforementioned bijective link between the Java object and the wrapped Objective-C instance. When the Java application invokes a method on the `UIAlertView` instance later, helper function `GET_ASSOCIATED_NATIVE` is used to retrieve the associated Objective-C object. It should be emphasized again that the JNI code shown above was also generated by our mapping tool.

Upcalls. The previous section dealt with the case where the application is making a downcall to iOS. In some instances the opposite happens: iOS makes an upcall to the application. This occurs frequently when iOS delivers events such as touch, timer, or sensor events. E.g., the `UIAlertView` calls the application whenever the user tapped on a button. In order to receive events, the application needs to register an appropriate callback, commonly referred to as a delegate. iOS makes use of Objective-C protocols for this purpose. Using the `UIAlertView` as an example again, iOS defines the protocol `UIAlertViewDelegate` that allows an application to respond to button-tap events.

What makes upcalls different from downcalls is the fact that for protocols the application needs to provide an implementation, not iOS. Mapping an Objective-C protocol to a Java interface lets the developer implement the interface, however, the resulting implementation only exists in Java space. As iOS has no knowledge of Java, a different kind of wrapper is needed. In the previous section we introduced a Java wrapper for an Objective-C object. Now it is just the reverse: what is needed is an Objective-C wrapper for a Java object. Figure 3 illustrates the reversal of roles.

From the perspective of iOS, a delegate needs to be an Objective-C object. The purpose of the delegate wrapper is to forward calls made by iOS to the Java application. The mapping tool can automatically generate these delegate wrapper for each Objective-C protocol based on the protocol's declarations. The following code excerpt shows the code generated for the declaration of `UIAlertViewDelegate`:

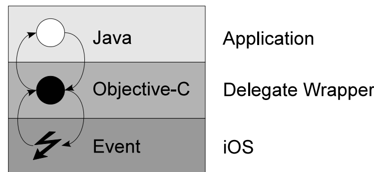

Fig. 3. Delegates in iOS

Objective-C wrapper for UIAlertViewDelegate
```
1 @interface UIAlertViewDelegateWrapper :
2                      NSObject <UIAlertViewDelegate> {
3    JAVA_OBJECT* delegate;
4 }
5
6      -(void)alertView:(UIAlertView *)alertView
7   clickedButtonAtIndex:(NSInteger)buttonIndex;
8 //...
9 @end
```

Class `UIAlertViewDelegateWrapper` is created by the mapping tool and conforms to the `UIAlertViewDelegate` protocol. The field `delegate` is a reference to the Java implementation of `UIAlertViewDelegate`. When the Java application sets a delegate, the above Objective-C class will be instantiated and registered with iOS. iOS will send the message `alertView: clickedButtonAtIndex:` whenever the user taps on a button. Not shown here is the implementation of `UIAlertViewDelegateWrapper` that is also created by the mapping tool. The implementation uses JNI to make the upcall to the corresponding Java method with `delegate` as the target.

A related problem occurs with upcalls made to regular iOS classes. E.g., class `UIView` offers a method called `drawRect:`. This method will be called by iOS whenever the surface of the `UIView` needs to be redrawn. An application can subclass `UIView` and override `drawRect:` to redraw itself. Here the situation is related to protocols in the sense that the subclass will be implemented in Java by the developer. As iOS has no knowledge of Java, a wrapper needs to intercept calls made by iOS to `drawRect:` and forward it to the corresponding Java implementation. The following code fragment shows the wrapper generated by our API mapping tool for class `UIView`:

Class Callbacks
```
1 // Objective-C wrapper
2 @interface UIViewWrapper : UIView
3 -(void) drawRect:(CGRect)rect;
4 @end
5
6 // JNI constructor for UIView
7 void UIView___INIT___(JAVA_OBJECT me)
```

```
 8 {
 9     UIView* obj = [[UIViewWrapper alloc] init];
10     ASSOCIATE(me, obj);
11 }
```

Class UIViewWrapper is derived from UIView to allow it to override method drawRect:. Not shown here is the implementation of UIViewWrapper that forwards calls to the corresponding Java implementation of method drawRect() via JNI. When a Java application instantiates a UIView, the JNI implementation of the UIView constructor will instantiate a UIViewWrapper instead. Wrapper classes will need to be generated by our mapping tool whenever a class features a callback such as drawRect:. This knowledge cannot be derived from the Objective-C header files and consequently needs to be provided via an external advise.

Memory Management. Earlier we mentioned the bijective relationship between the wrapper object and the native Objective-C object. In the following we focus on memory management issues as iOS and Java are governed by different memory management mechanisms. As explained earlier, iOS uses reference counting while Java makes use of a garbage collector. In terms of object lifecycle, a wrapper object cannot exist without the Objective-C object it wraps. Assume the example from the previous section. The application instantiates class UIAlertView that triggers the creation of an Objective-C object. iOS expects the application to release the Objective-C object in order to signal that it no longer needs the UIAlertView instance. This can be accomplished by registering a finalizer for the wrapper object. In Java, overriding method finalize() will cause the garbage collector to invoke this method just before the object will get collected. Making this method native allows again to inject code on the native level via JNI:

Finalization

```
 1 // Generated Java
 2 class UIAlertView {
 3     //...
 4     native protected void finalize();
 5 };
 6
 7 // Generated JNI
 8 void UIAlertView_finalize__(JAVA_OBJECT me)
 9 {
10     UIAlertView* thiz = GET_ASSOCIATED_NATIVE(me);
11     [thiz release];
12 }
```

Function UIAlertView_finalize__ will be called by the garbage collector just before the wrapper object is being collected. The associated native Objective-C object is retrieved via GET_ASSOCIATED_NATIVE followed by sending it the

release message. Note that the latter will not necessarily destroy the Objective-C object. If iOS itself holds a reference to the same object the reference count would not drop to zero therefore keeping the object alive. Whenever iOS no longer needs the object, it will send it the release message as well eventually destroying it. Therefore, an Objective-C object can outlive its wrapper but not vice versa.

So far we have discussed the relationship between a wrapper and the wrapped native Objective-C object. However, in terms of memory management one needs to consider relationship between wrapper objects as well. The unfortunate fact is that iOS will not always retain objects where one would expect. As an example, consider the protocol UIAlertViewDelegate that defines the API for a delegate used by UIAlertView. The UIAlertView allows the setting of a UIAlertViewDelegate instance that acts as a callback that iOS will invoke once the user has tapped on a button. Even though the UIAlertViewDelegate is set via a setter on the UIAlertView, the latter will not retain the delegate. This is the applications responsibility in iOS. Without arguing the merit of such an API design, the problem is that the following Java code using the Java-based iOS API would not work:

Setting a UIAlertViewDelegate

```
1 UIAlertView alert = //...
2 alert.setDelegate(new UIAlertViewDelegate() {
3     // React to button click of the UIAlertView
4 });
```

In Java it is common to use anonymous classes in these cases as shown in the code excerpt above. A Java programmer would expect that the delegate is saved internally. However, since UIAlertView is merely a wrapper that passes the invocation of setDelegate to the native Objective-C object, this will not happen in this particular case. The consequence is that the garbage collector will eventually reclaim the Java wrapper for UIAlertViewDelegate whose finalizer will release the underlying Objective-C instance. Since the UIAlertView did not retain the delegate, it will effectively be deleted even though it might still be needed. The result will be a segmentation fault since an invocation will be made on an object that does not exist anymore.

Parsing the Objective-C API one cannot identify such cases. The fact that an UIAlertView does not retain its delegate can only be known by studying the documentation. For the purpose of the API mapping, this knowledge needs to be passed to the mapping tool via an external advise.

4.4 Handling Exceptions

Although our API mapping tool tries to derive all necessary information from the Objective-C header files, in some cases this is not sufficient in order to generate the Java API and JNI code. Throughout the previous sections we gave several examples. The knowledge of such details is only contained in the documentation

that cannot be parsed by an automated tool. For this reason we have introduced the notion of *advise* that can be given to the mapping tool (see Figure 1 in Section 4.1). Someone familiar with the iOS API has to update the advise whenever a new version of iOS is released.

Advise can be provided to different components of our API mapping tool: the frontend that parses the header files as well as the two backends that generate code for the Java API and the necessary JNI code. During each stage of the translation process the relevant sections of the advise is consulted. The format of the advise is XML that complies to a particular schema. The following excerpt shows some of the information contained in the advise:

Advise

```
1  <replace pattern="//.*"/>
2  <replace pattern="__BEGIN_DECLS"/>
3  <replace pattern="__END_DECLS"/>
4  <!-- ... -->
5
6  <typedef java="boolean" c="BOOL"/>
7  <typedef java="byte" c="int8_t"/>
8  <typedef java="List" c="NSArray"/>
9  <typedef java="Map" c="NSDictionary"/>
10 <!-- ... -->
11
12 <class name="UIApplicationDelegate">
13   <selector
14    name="-application:didFinishLaunchingWithOptions:">
15      <arg position="1" type="Map<String,String>"/>
16   </selector>
17 </class>
18
19 <class name="UIView">
20   <selector name="-drawRect:" delegate="true"/>
21 </class>
22
23 <class name="UIAlertView">
24   <selector name="-setDelegate:" retain="true"/>
25 </class>
26
27 <class name="NSString">
28   <injected-method name="toString" modifier="public">
29     <signature>
30       <return type="String"/>
31     </signature>
32     <code language="c">
33       <![CDATA[
34         NSString* thiz = GET_ASSOCIATED_NATIVE(me);
35         return CONVERT_TO_STRING(thiz);
36       ]]>
37     </code>
```

```
38   </injected-method>
39 </class>
40 <!-- ... -->
```

The <replace> tags help the frontend to clean up the Objective-C header files before parsing further information (lines 1-3). These tags define regular expressions such as comment markers (line 1) that will be removed. The <typedef> tag defines basic type mapping rules. It can be used to define mappings for primitive types (lines 6 and 7) as well as class types (lines 8 and 9).

Advise that relates to a particular class is grouped with the help of the <class> tag. The methods for which advise is given are identified by the <selector> tag that references the usual Objective-C selector (lines 13, 20, and 24). Various XML attributes provide specific information about the method that cannot be derived by parsing the Objective-C header files. Attribute `delegate` (line 20) will trigger the generation of a delegate wrapper as explained in Section 4.3. Attribute `retain` (line 24) will make sure that a Java reference is held to the argument to prevent the garbage collector from reclaiming it as explained in Section 4.3.

The advise given for `UIApplicationDelegate` specifies that the type of the second argument can be narrowed to `Map<String, String>` (line 15). An example for code injection is shown for class `NSString` (line 27). This class is the iOS counterpart to `java.lang.String`. In order to easily convert a `NSString` instance to a `String`, the advise injects an additional Java method `toString()`. As can be seen in the advise above, besides the Java signature of the injected method the advise also provides the necessary JNI implementation. The latter is necessary since the knowledge of how to convert a `NSString` instance is out of scope for the mapping tool.

5 Example

In this section we make a side-by-side comparison of an iOS version of "Hello World" implemented in Objective-C and in Java using the API generated by our mapping tool. Xcode offers a tool called InterfaceBuilder that allows to define a UI using drag-and-drop through a graphical tool. InterfaceBuilder can also be used to create a boiler-plate iOS application with one click. The following code shows a programmatic version of "Hello World" to illustrate the API mapping:

iOS Hello World (Objective-C)

```
1 @implementation HelloWorldAppDelegate
2
3 -(BOOL) application:(UIApplication*)application
4     didFinishLaunchingWithOptions:(NSDictionary*)opts
5 {
6   CGRect r = [[UIScreen mainScreen] applicationFrame];
7   UIWindow* window =
8               [[UIWindow alloc] initWithFrame:r];
```

```
9   [window setBackgroundColor: [UIColor whiteColor]];
10  r.origin.x = r.origin.y = 0;
11  UILabel* label = [[UILabel alloc] initWithFrame:r];
12  [label setText:@"Hello World"];
13  [label setTextAlignment:UITextAlignmentCenter];
14  [window addSubview:label];
15  [window makeKeyAndVisible];
16  return YES;
17 }
18
19 @end
```

Below is the same application now implemented in Java using the Java API for iOS generated by our mapping tool:

```
iOS Hello World (Java)
1  public class HelloWorld extends UIApplicationDelegate {
2
3    public boolean didFinishLaunchingWithOptions(
4            UIApplication app, Map<String,String> opts) {
5      CGRect r =
6            UIScreen.mainScreen().getApplicationFrame();
7      UIWindow window = new UIWindow(r);
8      window.setBackgroundColor(UIColor.whiteColor());
9      r.origin.x = r.origin.y = 0;
10     UILabel label = new UILabel(r);
11     label.setText("Hello World");
12     label.setTextAlignment(UITextAlignment.Center);
13     window.addSubview(label);
14     window.makeKeyAndVisible();
15     return true;
16   }
17 }
```

As can be seen by direct comparison of the Objective-C and the Java version of the application, someone knowledgeable with the iOS API will immediately understand the Java implementation. Besides retaining naming conventions from the iOS Objective-C-based API, the Java version also demonstrates several mapping challenges discussed earlier: mapping of structs (`CGRect`), protocols (`UIApplicationDelegate`), and mapping of data structures to their strongly-typed J2SE counterpart (`Map<String,String>`).

6 Conclusion and Outlook

Apple favors the use of Objective-C for developing iOS applications. The purpose of our work is to give developers more freedom by offering Java as an alternative programming language. This paper focuses on the API mapping that is necessary

to expose the native Objective-C-based API in Java. Because of the sheer size of the iOS API our goal is to automate this process as much as possible. The ideas introduced in this paper have been implemented and released under an Open Source license. iOS app developers have successfully used our tool to develop Java-based apps and publish them on the Apple AppStore.

In the future we plan to focus more attention on the heuristics that lead to natural looking Java API. The heuristics currently used in our tool are based on the way Apple has designed its API (e.g., the fact that function names that operate on certain structs are prefixed with the name of that struct). We plan to investigate API outside the iOS ecosystem to derive a more general set of heuristics. Ultimately our tool should be able to create Java API for other third-party libraries.

Acknowledgements. We are greatly indebted to Paul Poley and Panayotis Katsaloulis for their invaluable insights and support.

References

1. SWIG - Simplified Wrapper and Interface Generator, http://www.swig.org
2. Adobe Systems. PhoneGap, http://wiki.phonegap.com
3. Beazley, D.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, vol. 4, p. 15. USENIX Association, Berkeley (1996)
4. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (October 2010)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (October 1994)
6. Hudak, P.: A semantic model of reference counting and its abstraction. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, pp. 351–363. ACM, New York (1986)
7. Kochan, S.: Programming in Objective-C, 4th edn. Addison-Wesley Professional (December 2011)
8. Krasner, G., Pope, S.: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System (1988)
9. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley Pub. Co. (April 1999)
10. Lougher, R.: JamVM, http://jamvm.sourceforge.net/
11. Object Management Group. Common Object Request Broker Architecture, CORBA/IIOP (2004), http://www.omg.org/technology/documents
12. Puder, A.: Running Android Applications without a Virtual Machine. In: Venkatasubramanian, N., Getov, V., Steglich, S. (eds.) Mobilware 2011. LNICST, vol. 93, pp. 121–134. Springer, Heidelberg (2012)
13. Smaragdakis, Y., Batory, D.: Implementing Layered Designs with Mixin Layers. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 550–570. Springer, Heidelberg (1998)
14. Xamarin. MonoTouch, http://docs.xamarin.com/ios