Mobile Apps Multi-Platform Design Pattern
Featuring Translator for Interactive Animation Components

JUAN R. REZA, Research Consultant

ABSTRACT

The Android and Apple platforms for mobile apps are based on Java and Objective-C, respectively. Each platform runs user-interface frameworks suited to their language and operating systems. Frameworks are Cocos2d and Android libraries, supported by iOS and Linux/UNIX, respectively.

Some of the techniques for translation of code from one language to the other are presented in this paper. It is the animation and interaction through platform-specific infrastructures that presents a challenge to the goal of developing one code based for both platforms. The design pattern defines the translator-based approach in detail after listing some of the trade-offs between that and a major alternative, the 3rd-party cross-platform framework approach. This pattern is largely expressed in terms of two specific languages and platforms for purposes of clarity. However, the concept should be easy to see as generalizable to other languages. The solution recommended by this design pattern is expressed as a structure and process. The Structure section depicts the architecture of a cross-platform app that results from applying the translator during development. The Process section describes the development process steps culminating in that structure.

Some of the coding techniques of the translation approach are presented in summary form. Work with that approach is in-progress for a commercial-grade set of apps designed for iPad and certain Android devices. The approach is proving to be practical. This design pattern captures the points to consider when choosing an approach to cross-platform development, particular when the application will be dependent on profoundly different hardware and platform-specific frameworks as is the case in interactive animated mobile apps.

Keywords
software design patterns, Mobile apps, Cross-platform, Objective-C, Java, Code generation, j2objc, Android, iOS, cocos2d, storyboard, UIViewController, mock objects.

1. The Pattern

1.A Introduction

The Android and Apple platforms for mobile apps are based on Java and Objective-C, respectively. Each platform runs user-interface frameworks suited to their language and operating systems. The key frameworks of interest in this paper are Cocos2d and Android libraries and the platforms iOS and Linux/UNIX, respectively. This paper does not go into how to develop the animation and user interaction such as gestures. It is concerned with cross-platform development when a large part of the design consists of platform API-specific code that would naively require duplicate development.

Developers recognize the concerns of app development that are peculiar to mobile apps and have offered up certain design patterns for them [1] [2] [3] [4] [5]. These experiences and recommendations express concern with the end behaviors of apps and not about their development with multiple platforms as a concern. The question of best practices for cross-platform development is far from being settled.

Design Patterns represent recurrent needs, problems encountered in an effort to fulfill those needs, and solution concepts that stand up well even as technology evolves. As such, the description of a pattern should be independent of any one language or hardware. The design pattern presented here, however, is expressed in terms of two languages based on a similar object-oriented philosophy. To do otherwise would make every concept so abstract as to be difficult to follow.

The pattern has been recognized during a time of a rapidly evolving set of languages and devices. It is a moving target. It describes particular tools and products rather than attempting to elevate all of its concepts to an abstract level that might be more robust over time. In the interest of presenting an immediately practical pattern, the comparison of alternative solutions is specific to the narrow set of technologies.

We see an effort to identify the components that could be developed once for both technologies. The main alternatives are: (a) create a 3rd technology that can work in both systems; (b) create a translation or adapter mechanism to allow
development on one technology that is compatible with the other. This later alternative is the approach taken by the design pattern presented in this pattern.

In keeping with the evolutionary pattern of competing technologies described above, this paper examines some of the trade-offs faced by software development teams. These trade-offs are applicable to this point in time and will change as the products mature or fall out of favor in the community. Trade-offs are also subject to opinions. One analyst might see a defect or risk in one approach that is true at its present level of maturity, while another analyst might discount that deficiency as unimportant because of the expectation that the technology will soon mature. That view would incur some risk but offers the hope of putting the resulting product ahead of the crowd. Another set of trade-offs is based on whether you are more comfortable with relying on third-party tools and components to base your investment on, or whether you want to keep stronger control, ownership, and knowledge in your own hands; to avoid depending on resources that you do not control. The conservative trade-offs express the attitudes that would be supported by the design pattern of this paper. The choice between translation and third-party cross-platform framework might not be obvious because of the mix of tradeoffs. For this reason, this design pattern includes some comparisons between these two approaches showing the conditions in which this pattern would be the more favorable choice.

Some of the techniques for translation of code from one language to the other are presented in this paper. The design pattern defines the translator-based approach. The Structure section describes the architecture of a cross-platform app that results from applying the translator during development. The Process section describes the development process steps of the pattern. For consistency and simplicity in this paper the two approaches are informally called (a) the translation approach and (b) the third-party cross-platform framework approach. A separate design pattern could report on experiences with the cross-platform framework approach. That should be done by developers using that approach for a comparable objective.

1. B Audience

The design pattern described in this paper is intended for …

- People interested in mobile app development.
- Developers and software architects interested in approaches to the problem of cross-platform development and to learn about other practitioners experiences.

2. Context

- Each of two mobile platforms supports one of the programming languages. Each platform also has its own User Interface (UI) framework that is design to take advantage of platform features.
- A significant investment is already committed in developing apps for one of the two platforms. Or, the objective of the development project is to design an app for the two platforms up front.
- Developers available to the project are more familiar with just one of the platform languages.
- A cost-effective third party language and framework is available. (This pattern is applicable if a translator is available.)

3. Intent

In developing a mobile app, we need to develop the app for two classes of mobile platforms in one language as much as possible so as to avoid duplicate development of functionality. Our objectives here are …

- To avoid duplicate implementation of identical functionality where possible on different platforms.
- To reduce dependency on numerous plugins and frameworks.
- At project conclusion, the software developed to enable cross-platform development should be highly re-usable.

We don’t expect a translator or cross-platform tool to hide from us all the platform-specifics so that we would code for one platform in complete ignorance of the other. We expect the tool to let us use our knowledge of both platforms well and without excessive hand-holding of the tool itself.
4. Motivation (The Problem)

- There is a cost to developing and managing more languages, development tools, and application variants for two mobile platforms. Generally it is difficult to effectively develop a single product for two different platforms that achieves identical behavior.
  - Manually re-implementing, or implementing in parallel, a code base for two platforms almost guarantees that the app will behave differently in unintended ways that is not due to platform differences that made identical behaviors impossible.
  - In the case of platform-specific UI frameworks that are based in two different languages, this fact could drive the re-implementation of all the business and integration logic just to accommodate the UI component. The 3rd-part cross-platform language could be an approach to this particular problem but presents other difficulties.
  - There is a cost to maintaining staff or contractor skill sets necessary to support each tool and variant. The ability to work with numerous software tools and languages is valuable to a company and on a resume. But there is a trade-off for diffuse attention. Skill in a few languages can allow the same individual to achieve higher proficiency and nuanced design knowledge than when short attention is given to numerous subjects or in reaction mode.
  - Coders and software designers often know and love either the Java frameworks, or the C legacy languages, or Objective-C but not all equally well. Frameworks in these languages pose an additional learning curve

- Based on past experience with interoperable cross-platform products, especially if not mature, we are skeptical about committing large resources to a 3rd-party framework of that kind while there is a viable, less exotic, simple and free alternative.
  - Available IDEs for app development work for one platform (out of the box), thus requiring development work to be performed outside the IDE or with extensive customization with 3rd-party plug-in frameworks. For example, you might have to run scripts manually for certain steps such as creating a project for the IDE which is normally an IDE function.
  - All-in-one code cross-platform solutions often work for the normal situations, but when they fail or generate incomplete or buggy code, you cannot modify or apply fixes easily or in round-trip engineering mode.

5. Solution Summary

- Choose the primary language and the secondary language. Most of the development work will be in the primary language and code in the secondary language will be generated by translation of the primary language code.

- Allocate the functional requirements of the app into three components (groups of components) comparable to the 3 tier architecture of enterprise applications.
  1) BI: Business and Integration\(^1\) (aka Persistence) tier is the “intelligence” of the application. It can be coded once in one language and should be able to work on any mobile device, comparable to the controller and model. (i.e. it “should” be easily ported or completely interoperable because it does not have any dependency on platform-specific UI capabilities.)
  2) UI: User Interface logic that is very specific to the type of mobile device platform, comparable to the web tier. (i.e. you “have to” use the device-specific UI API at some point.)
  3) Bridge: The “bridge” logic is designed to decouple the BI and UI components, comparable to a web protocol.

- Automatically generate code in the language of the other platform for the BI and Bridge.

---

\(^1\) BI: A more-general term would be “application” and integration, but that would confuse with artificial intelligence AI. Many apps are not exactly “business” functions such as games, computational, personal data, news-like functions.
• Implement the UI for each platform, using the Bridge to communicate with the BI components.

In Figure 1 the Bridge Logic component represents very loosely a collection of classes. It is not a direct realization of the GoF Bridge pattern [6]. The next section reveals more of the Bridge classes do. Figure 3 will expand the details.

![Figure 1. Simplified view of app architecture using the translation approach to cross-platform development.](image)

5.A Process

The development process based on automatic translation of app code is an important part of the design pattern. The steps for how the artifacts of the solution are created and used are listed here at a very granular level for clarity.

P-1) **BI**: Business and Integration (aka Persistence) tier which can be coded once in one language and should be able to work on any mobile device after translation. Implement these components in one language; the primary language. In step P-4 these component will be automatically translated into the secondary language and run on the other target platform.

P-2) **Bridge**: The “bridge” logic is designed to decouple the BI and UI components. Implement this logic in one language (as in the BI step) and generate the other. The Bridge logic is concerned with general functionality of a UI and behaviors rather than application-specific requirements.

a. As part of the Bridge component, create and use **Mock components** for the UI in test mode.

b. Manually re-write some of the Bridge code for the other platform, using the translated Bridge logic as a reliable starting point for platform-specific API calls. This should be uncomplicated since the functions are well defined and generic in nature.

c. The concern of the Bridge classes is to support translation. The Bridge is not intended to encapsulate the hardware/OS differences among mobile apps. The Bridge would not, for example, compensate for the absence of capability to detect a particular gesture or motion on one of the platforms.

P-3) **UI**: User Interface logic that is very specific to the type of mobile device platform. Implement these components in both of the platform languages “manually.” This manual step might be done using the automation available in the platform’s favored IDE. Write these components to call and provide call-backs to the Bridge logic.

P-4) **Translate**. Use the original BI and Bridge code for testing and deploy to platforms as-is (assuming translation is accurate). Although we call this code “business” logic it can refer to the “business” activities of a highly interactive animated game; its decisions, behavior responses, and such.
P-5) **Test original:** test the functionality of the fully integrated app using the original BI and the original-language Bridge code.

P-6) **Test translated:** Test the functionality of the fully integrated app using the translated BI and Bridge code with the UI which is native to the other platform.

P-7) **Deploy** the successful tested sets for each platform.

Figure 2. Development process for a cross-platform app using the translation approach.

**Notes.**

UI event handlers: the naïve coding practice involves the coding of behaviors closely to the objects of the UI. For example, the response to “touching” a Sprite might be coded directly and using platform-specific (convenience) methods. That is fine given that you are developing for the host platform only.

This solution, however, recommends passing events from the platform-specific UI, through the Bridge logic, and letting the BI code make decisions about what the Sprite should then do. Send control information via call-backs to the UI. As an example, instead of attaching clock events to a Sprite event-handler to make it glide across the screen, consider having the BI logic make positioning decisions more granularly, even to the point of processing the 60Hz clock notifications from the UI.

This approach assumes that an appropriate trade-off will be made between maximal utilization the GPU of the device for high-speed action (as in games) versus development concerns such as rapid reliable implementation of complex functionality. The impact of an extra layer of calls and call-backs involve very few executable statement, so should have little impact overall.

5.B **Structure**

The cross-platform component architecture of the app will resemble the diagram Figure 3. As mentioned earlier, this design pattern refers to the specific languages and platforms, Java-based and Objective-C –based. It would be confusing to use completely abstract terminology such as language 1 and 2. It is left to the reader to see these languages as representatives of a generalizable pattern. The use of specifics is also necessary to discuss examples of translation issues and techniques as shown in the Implementation section 8.
There are features of this particular pair of languages, Java and Objective-C, and their respective UI frameworks that invite reliable automatic translation. They are both single-inheritance. Their instance and a class members have analogous characteristics. Their core language and their UI frameworks have many comparable classes, differing in name and details, that are obvious candidates for creating bridge classes for those correlates. The design philosophies seem very compatible, as a matter of opinion at least, in other subtle ways.

Other languages, if paired with one of these, might not be so conducive to this bridging approach. C++ multiple inheritance might pose difficulties with translation to or from, for example.

![Diagram](image)

**Figure 3.** Component structure of completed cross-platform app under the Translation approach.

**Legend.**
- Blue bi-directional arrows indicate runtime communication (method calls) in the app.
- Red arrow indicates how components were created (by the translator).

The UI Action-Initiating classes and the UI Event-Handler classes written in the first platform (on the left in Figure 3) are of two kinds.

Those that will be part of the code base of the first platform, and those that will be replaced by equivalent code in the second platform. Of these, the second platform either has the equivalent class as part of its native framework, or an equivalent bridge class must be written. See the example of the Sprite or NSSprite classes in section 8.

### 6. Consequences

By planning the project for dual-platform implementation, we can avoid coding and design decisions that would make later porting more difficult. Quality-assurance engineers can avoid duplicate effort by testing the outward behavior of graphics separately from the computational decision, the “intelligence”, that drives the graphics and respond to events.
6.A Comparison of advantages and disadvantages

Table 1 suggests some of the trade-offs to be considered in choosing a cross-platform strategy for app development. The list of ways that the approach can impact the development process are not intended to be mutually exclusive. They are separate perspectives on the features of each approach.
Table 1. Comparison of advantages and disadvantages (in no particular order).

<table>
<thead>
<tr>
<th>Likely Impact of Approach Considered</th>
<th>Translation</th>
<th>Cross-Platform 3rd-Party Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Eliminates <em>all</em> need to translate code across platforms</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>2. Eliminates need to translate the interactive and animation components, mostly if not all.</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>3. Complete app can exist in one language; no calls to native language</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>4. Does not require any manual translation of code not covered (by translator or framework)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>5. The original code of an app is written in the language more natural to its native platform. (in other words...) An app does not have to be written in a language extension that is foreign to both intended platforms.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>6. Code has full access to the features native to the platform it is designed for. (e.g. a feature that uses the hardware.)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>7. There is no risk of a bug in the 3rd-party framework impacting the deployed app</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>8. Development and maintenance of the app requires staff with knowledge of n languages/extensions/frameworks</td>
<td>N=2 (adv.)</td>
<td>N=3 (disadv.)</td>
</tr>
<tr>
<td>9. If the cross-platform implementation (by translator or by framework) is behaving in unexpected ways, or not the same on both platforms, you can (a) see the effective code, and (b) fix it for one platform, and (c) any work-around for one platform does not affect the app working on the other platform (no re-testing on other platform).</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>10. If one vendor introduces a desirable feature on their platform and, via their native language, you can use it immediately without waiting for availability of support via your approach on the other platform.</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

The rendering behavior of controls and animated objects on the screen is under the control of the UI framework. (The “business” logic tells UI what to do, and receives semantically complete events, and the UI digests lower level details.) The UI, working closely with the platform GPU, does a lot of the work of making the Sprites move smoothly and detecting collisions, for example. However, more of the “cognitive function” of a game is under the control of the BI components, the position and trajectory of characters is known. So, it may be unnecessary for the game logic to provide collision notification call-backs: the BI “knows” where things are and will be, even when positions are changed by touch events. That can be an advantage. However, the trade-off might be moving some runtime code away from the GPU.

Given the previous trade-off, if it turns out to be necessary to move some action functionality back to the GPU, you have the code that is specific to the device having performance problems. [Contrast this with reliance on a single cross-platform framework.

---

2 GPU: graphics processing unit.
6.B Comparison of development tools

The infrastructure necessary to the two approaches is considered here. In one respect, the cross-platform approach using the Cocose2d-X as an example, appears to impose more tooling overhead than the translator approach.

The special tools for using the translation-based approach are listed along side those needed at this time for the evolving 3rd-party approach. But there is no one-to-one comparison of any specific tools. The impact of these infrastructure tools and dependencies is left to the reader to interpret. The sheer number of special tools for the cocos2d-X platform [7] [8] may be daunting compared to the one tool for the j2objc translator. On the other hand, someone already enthusiastic about the beauty of a single cross-framework might dismiss as unimportant these “one-time” infrastructural overhead items.

3rd-party cross-platform framework approach:

- Android NDK (Native Development Toolkit)
- Eclipse C++ Plug-in for MacOSX platform developed by UNKNOWN
- $NDKROOT/ndk-build shell tool developed by UNKNOWN
  - allows you to integrate external C/C++ libraries (like Cocos2d-x) into Eclipse ...
  - To communicate with the “legacy Java Android architecture”
- create-android-project.sh is required to build a project outside of Eclipse.
- classes created specifically, or in a specific way, to accommodate aspects of the cross-platform framework.

Translation approach:

- J2objc – the open-source translator from Google [9].
- classes created specifically, or in a specific way, to accommodate aspects of the cross-platform framework.

7. Applicability

Use the Apps Featuring Interactive Animation pattern when . . .

1. You plan to implement an app on both the Apple Objective-C (iOS) platform and the Java Android platform.
2. You have a completed app on the iOS platform and wish to port it to Android and maintain it on both platforms.
3. You have a completed Android application and, given that you have considered the trade-offs, you wish to re-implement it for the iOS platform (say with new features) and continue maintaining it on both platforms.

Note also...

A mature Objective-C to Java translator from Apple is also available for consideration [10] [11]. It was not evaluated for the purpose of this paper. However, if it serves the purpose of your project, the design pattern presented in this paper could apply except with all references to iOS and Android reversed. According to the documentation that translator may have more capability to convert UI-level classes than the j2objc translator for the purpose taking advantage of device features.

Alternatively, you may have considered total reimplementation in C++ to use cross-platform framework cocos2d-X [7].

8. Implementation

This section presents how to implement this design pattern with several examples of the bridge classes. It is assumed that a number of comparable classes will be developed for an actual app. The business and integration classes are expected to be translated with no problems due to compilation-compatibility with the respective UI frameworks because they will only interact with the Bridge code directly.

These example represent the basic kinds of capabilities that bridge classes provide. One obvious kind of bridge class is the mock class. This allows the original business-integration component to interact with the UI through the bridge during testing (without the actual UI). Another kind of bridge class implements a class in the first language that is a direct replacement in the second language. This bridge class functions like an actual class in the second language that is not
available in the first language. It interacts with the UI of the first platform, and its correlate (which is a native part of the second language) is simply used directly by the translated BI code.

An important feature of the j2objc translator is the way it generates the name of an Objective-C class. Given a java class such as MySprite (Listing 1), the translator creates the header, implementation, and all references to them according to the prefix you can specify. So MySprite (java) would generate as CCMysprite for the prefix CC. The way to take advantage of this prefix is to create java packages within the BI and Bridge component named in correspondence to the Objective-C prefix you intend for those classes. This is particularly useful for generating the mock classes. For example, to create a mock CCSprite, create a java class with cc in the path such as com.company.myapp.cc.Sprite. The same can be done for various other cocos2d classes. Naturally, since you are developing primarily in Java, the Bridge will need to support the Android API corresponding to cocos2d as well. If your bridge classes separate their logic and API-specific code well, the UI-specific code can be kept clean and maintainable.

### 8.A Mock classes in the Bridge

The NS package of Objective-C is likely to consist of only mock classes. As an example, consider using a static method in Java as defined in Listing 1. While your java code can call Log(“answer to life, the universe, and everything.”), your automatically generated code everywhere will call the appropriate Objective-C equivalent, NSLog: @”message”.

**Listing 1. Example cross-platform friendly java class Log**

```java
1. package appPlatformBridge.ns;
2. import static appPlatformBridge.ns.Log;
3. /**
4. * usage: <br>
5. import static appPlatformBridge.ns.Log.Log; <br>
6. call as {java}: Log("answer to life, the universe, and everything.") <br>
7. generates call (Objective-C): <br>
8. NSLog(@"answer to life, the universe, and everything.") <br>
9. */
10. public class Log {
11.   public static void log(java.lang.String format) {
12.     System.out.printf(format.toString());
13.     System.out.flush();
14.   }
15.   public static void log(java.lang.String format, Object... text) {
16.     System.out.printf(format.toString(), text);
17.     System.out.flush();
18. }
```

### 8.B Bridge classes for deployment in the first platform

Another important package is the CC-prefixed set of cocos2d classes.

Your Bridge component might include a Sprite class. Unlike the previous example of Log, NSLog, the Bridge Sprite class is not just for mock purposes. As Java, it will contain Android-specific code as well as XML. The java may look like...

**Listing 2. Example cross-platform friendly class for direct use in Android, mocking direct replacement in cocos2d.**

```java
1. package somewhere.cc; // java package cc is a bridge-tier class mapping to CC cocos2d classes
2. // Mock the second platform Sprite to support bridge class Sprite in the first platform
3. public class Sprite {
4.   ...
5.   private void update() {
6.     xSpeed = myAccelerateFormula();
7.   }
8.   public void onDraw(Canvas canvas) {
9.     update();
10.    canvas.drawBitmap(bmp, x, 10, null);
11. }
```

3 “First platform” refers to the java/Android platform and framework in our example consistent throughout this paper. The translator generates “second platform” code (Objective-C and cocose2d) from the code in the first platform.
12. **package** somewhere.else.ap; // java package ap is a business-integration tier class
13. // First platform implementation of Sprite, capable of translation to second platform
14. **public class** MySprite extends Sprite {
15.     **private void** update() {
16.         myDecisionFunctionality();
17.         super.update();
18.     }

The reason we go to the little trouble of creating a Sprite class for the Java-based platform and then extend from it in MySprite is to allow for automatic translation into the cocos2d-based platform which has an intrinsic CCSprite class.

Using the Bridge in Java, the calls to Sprite methods that resemble…

```
Lvalue = Sprite.method(args);
```

will be translated to Objective-C as …

```
Lvalue = [NSSprite method]:args;
```

However, there is no need to translate your java cc.Sprite class into Objective-C since NSSprite is a direct replacement.

### 8.C Some other platform considerations

We identify a number of coding practices that help the converters. For example, it is not sufficient for a converter to see “BOOL” if its target language could convert the element to either Boolean or boolean, or to **int** if the source language has valid statements like **BOOL** x = 0;

An important consideration is when one platform provides a powerful hardware GPU (Graphics Processing Unit) while the other does not or is not as easy to access. As a result, certain graphic or Sprite transformations can be implemented with simple calls while in the other the feature might be hard to implement or become degraded by performance problems.

Computation of positions, timing, rotations and so forth are candidates for allocation of features to common code in the business-integration tier. The Bridge is not be responsible for functionality just because it is “common”. Its responsibility is to provide a simple mechanical set of replacement and mock classes. This might seem contrary to coding graphics activities that are geared to coding close to where the action is. Performance considerations should lead to more early computation rather than computing graphic animation on the fly in all cases.

Platform awareness such as clock rate is not the concern of the bridge. The business tier uses the bridge solely as a mechanical intermediary supporting translation. Configurability of the application to account for platform hardware and product features rests with the business logic. This includes classes unique to a platform that query and modify platform state for which the bridge will only provide compilation-compatible bridge classes.

When it is necessary to write UI code separately for each of the two platforms, it is easy to lose sight of the common conceptual steps. For this purpose, this design pattern calls for more granular methods, corresponding to intuitive “steps”. This will help in either case of using a converter, or manual re-implementation.

### 8.D Implementation planning

The foregoing examples reveal that each of the two-character package names/prefixes, tends to host bridge classes of a certain kind. Those destined to be translated and used directly in both the first and second platform should be in a package that can be managed that way. Those classes design to serve as mocks for the first or second class can be similarly isolated and used in build procedures together. The details will vary for a project or development group and will certainly vary for any language combination other that the two used as examples throughout this paper.

---

4 “ap” package is just for example, unlike cc, ns, and other 2-character packages names that correspond to CC, NS, etc.
5 These examples incidentally use static methods but the same concept applies to instance methods.
9. Conclusion

The design pattern described in this paper, while largely in terms of two specific platform languages defines an approach to accomplishing the stated intentions. It shows a way to avoid duplicate implementation, reduce dependencies, provide control over tools and frameworks for translation, allow manual rewriting of select platform-specific components, and isolate classes good for re-use. Comparisons to the main alternative on these points should allow early design decisions about which approach is best for the development of interactive animated mobile apps.

Mobile apps that feature interactive animated graphics, sound, and images must be implemented differently on the two prevailing app platforms. However, much of the functionality can be developed in one language and ported relatively painlessly to the other language. Experience shows that Java-to-Objective-C and Objective-C to Java converters are useful but not perfect. The details of coding style can help the converted to “understand” the intention of certain statements. The allocation of functions to platform-specific packages can also help to reduce re-implementation and late discovery of platform issues that could require moving code and re-implementation.

10. Question-Answer

This section is introduced as a supplement to the traditional form of design patterns. The questions arose from discussions during the peer-review process.

QUESTION 1.
Regarding Table 1. Line 8: Does the developer need to know 3 languages by not using this pattern?

ANSWER
Using this pattern, No: two languages. Using the alternative of a 3rd party cross-platform framework in addition to the two platform-native languages, yes. As an example, take Cocos2d-X. This cross-platform framework is a language embedded in Java. As such you need to know Java. The classes and methods it provides follow the original ones in Objective-C Cocos2d, thus 3 languages. Currently, any features that cocos2d-x does not implement, which has limitations and platform differences, will have to be accommodated by skill in either or both of the native UI languages.

QUESTION 2.
So some Bridge logic is coded in the primary source language and then translated to the target language. Is that dependent code of the secondary language then changed to optimize for that platform, as the process section (5.A step P-2 b.) seems to imply? If so, what happens when you change code in the primary language? You have to change the dependent secondary code again. It seems that generated and modified code gets “clobbered”.

ANSWER
The primary language code that implements Bridge components is often emulating components that are built-in or in development for the secondary platform language. So there is no need to translate them except for creating mock elements for testing, say. In that case, no secondary language code gets iterated.

Functionality that is not already available in the secondary language and cannot be produced by automatic translation needs to be implemented manually. The translation of the Bridge component written in primary language can be used once as a starting point that serves to define what the primary language expects. Naturally, if that definition changes, the derivative code in the secondary language would need to be changed, just as other parts of the primary code would have to change if they are dependent on it. The developer would not discard the previous version of secondary code, necessarily, but use what can be preserved. The implementation should anticipate the possibility of iteration.

This process should never be interpreted in a way that manually written code has to be done over again because of the translator.

QUESTION 3
Regarding section 8.A and 8.B, why not show source and target code side by side?
Listing 1 shows an implementation of two overloads of Log() written in Java that emulate the Log method which is a built-in in Objective-C. As such there is no reason to show an implementation of Log in Objective-C. As for how to call Log, line 6 gives an example of the call.

Similarly, Listing 2 shows an implementation of part of Sprite. Sprite is available in Objective-C. This Java implementation of Sprite lets the Java (primary language) behave in an equivalent way as its cocos2d counterpart.

References


