Sensing Mobile Phone Interaction in the Field

Florian Lettner
Department of Mobile Computing
University of Applied Sciences Upper Austria
Softwarepark 11, 4232 Hagenberg, Austria
florian.lettner@fh-hagenberg.at

Clemens Holzmann
Department of Mobile Computing
University of Applied Sciences Upper Austria
Softwarepark 11, 4232 Hagenberg, Austria
clemens.holzmann@fh-hagenberg.at

***Abstract*—Sensing how people interact with their mobile phones under real-world conditions can be an expensive and time-consuming task, especially if data from a large number of users and over a long period of time is needed. However, in order to improve the utility and usability of mobile applications, field studies are often considered to be more suitable than laboratory evaluations. In this paper, we present an innovative approach towards the automatic and transparent sensing of mobile phone usage, which is based on the background observation and analysis of user interactions with a mobile application under real-world conditions. We have implemented a software framework for this purpose, which can easily be added to an Android application, in order to record how users interact with it and transmit the acquired data to a server. We evaluated our framework with regard to its performance and memory consumption on the one hand, and by adding it to an application in the marketplace on the other hand. The results from about 300 users over one week showed that the framework runs stable and with very low resource demands.**

***Keywords*-Mobile interaction, software framework, aspect-oriented programming, field evaluation of usability.**

I. Introduction

Usability evaluation on mobile applications is a complex task, and the collection of larger amounts of representative usability data by conducting laboratory-based tests is often not feasible. However, Duh et al. [1] found out that authentic test data will not be provided by laboratory-based evaluation, because of the fact that real-life conditions cannot be simulated sufficiently well. Instead of also carrying out field studies for high-level prototypes and final applications, usability experts often limit their evaluation to the early stages of interface design. According to Duh et al. [1], performing usability tests based on low-fidelity prototypes, where test participants attempt to operate a mobile application under real-life conditions, or introducing distraction tasks, where users are confronted with actions and tasks that are not related with the tested mobile application [2], are state-of-the-art for mobile usability evaluation though. One of the main reasons for it is that alternative evaluation methods hardly exist and that field studies are often too expensive to be applicable [3], [4]. However, the option of emulating real-life context and user distraction only provides a coarse and limited approximation of a natural environment.

In addition, Morris et al. [5] point out that mobile software engineering is a process that is highly agile and incorporates several macro- and micro-iterations during the development process. Basically, software engineering is the same for desktop, web and mobile applications. Macro-iterations in the design of a software can follow a common software development model such as the waterfall model (i.e. requirements, design, programming, testing and deployment). However, early user acceptance tests due to instant feedback from mobile markets, for instance, give developers the opportunity to provide quick fixes or change minor product components. This leads to faster improvement and faster changes of mobile applications without major significant modifications to the application itself, which is referred to as micro-iterations. However, performing supervised field studies as well applying general usability evaluation methods for fast software iterations is infeasible for companies, because they are too expensive and time-consuming [3] to be applicable for every iteration of the software version.

For these reasons, the aim of our work is to facilitate the field evaluation of high-fidelity prototypes and applications in the post-purchase phase.

A. Contributions

Due to a lack of tool-based usability evaluation methods [4], we present a method as well as a framework for automatically sensing user interaction on mobile phones, in order to enable cheap and fast usability evaluation of mobile applications. Additionally, developer requirements for "convention over configuration" – a method for dependency injection – are presented to further improve the efficiency and reduce the costs for field tests. Dependency injection refers to a software design pattern that tries to minimize dependencies between application modules to enable, for example, the reuse of certain program modules. With regard to the presented software framework, this means that code written by application developers should not have to be manipulated or changed in order to use the framework, or in other words, that the evaluated mobile application should not have to have knowledge of the framework which senses the user interaction.

Our framework is based on a generic concept that makes

it possible to transparently monitor a user's operational behaviour by injecting source code into the life cycle of an Android-based application at compile time. The presented concept for source code injection, for which we use aspect-oriented programming, also reduces the developer's effort to integrate our software framework, as no manipulation of the mobile application's source code is required. In the following, we will first present the theoretical background and the concept of our framework in Section II. It will cover dependency injection based on aspect-oriented programming for Java, and outline how the framework interacts with host applications by adding source code at compile-time.

The proposed software framework for usability evaluation gives developers and designers of mobile applications the ability to enhance the quality of their software by automatically sensing how users interact with their applications under real-life conditions. In Section III, we will present the framework architecture and describe how it can be used for the automatic sensing of user interaction with an Android application, based on the observation of its life cycle. This section also provides details on the plugin-based architecture which has been developed in order to achieve a robust, flexible and scalable solution.

In section IV, we will finally provide facts and figures about the performance of our framework, including its memory consumption and network payload. This data has been collected during a feasibility study with a real application, in order to show that the framework does not influence the host application in a negative way.

II. Sensing Interactions with Mobile Phones

In order to reduce the effort required for carrying out field studies for mobile applications, we were searching for a solution to integrate third-party source code components to Android applications, which monitor the user interaction with these applications. This means that the users should not be observed by usability experts any longer, but they should instead be able to use the respective applications in their day-to-day work flow without any distractions.

Therefore, a solution for sensing user interactions with a mobile application was required, in order to provide valuable usability-related information which should eventually aid developers and designers in improving their mobile user interfaces. On the one hand, the overall goal was to meet the user requirements to automatically collect and provide usability data. On the other hand, the framework should also ease the developers' lives by providing possibilities for its fast and easy integration.

A. User Requirements

One of the most important requirements was to provide the option of collecting usability data in the field in order to support the evaluation of high-fidelity prototypes and applications in the post-purchase phase. The usability evaluation process should work transparently, so that the users' work flow is not directly affected by the proposed framework. In other words, the framework should not require any direct interaction with the user, which might disturb his or her interaction with the host application. Thus, users should be able to use their mobile applications naturally and under real-life conditions, where it is more likely that they are sitting in the train or walking down a street than spending hours in front of a big screen in a silent office[6].

B. Developer Requirements

Simply providing a tool-based solution for conducting field tests on mobile phones does not necessarily means that they can be carried out efficiently. According to Kaikkonen et al. [3], field studies often require too much time to be applicable for commercial solutions, but there is another issue: Higher preparation times and costs arise from the the fact that the integration of usability frameworks requires a serious amount of time and that developers require training which can be too expensive.

Therefore, the major requirement regarding software developers is to decrease the number of decisions they have to make without losing flexibility when using the proposed framework. Therefore, a method was required to automatically add code at predefined locations within an application, such that developers do not have to manually call framework functions at certain points in their code.

C. Implications

To meet the majority of identified requirements, decisions for the use of the right technologies are crucial. For simplicity reasons, we decided to go for the Android platform. On the one hand, Android is a young and aspiring platform, which – as a market leader with a market share of 36% in the first quarter of 2011 [7] – offers many tutorials, simple development concepts and a good documentation. On the other hand, Android is a very open platform, and the source code is freely available which eases more difficult development tasks by giving insight into the operating system architecture.

Regarding dependency injection, we decided to go for aspect-oriented programming (AOP). AspectJ, an AOP framework for Java[1], can be used for the development of a framework that is added to any Android application transparently, so that the host application does not have to have knowledge about the presence of the framework. Hence, application code written by developers does not have to be manipulated or changed in order to use the evaluation framework. In terms of dependency injection mentioned in Section I, the proposed framework works independently and does not have to be managed by the host application. With AOP, it is possible to define software modules that can be

[1] http://www.eclipse.org/aspectj

added to applications after they have been compiled or even at runtime. Moreover, the proposed framework can be added automatically to any host application that is built upon the Android life cycle.

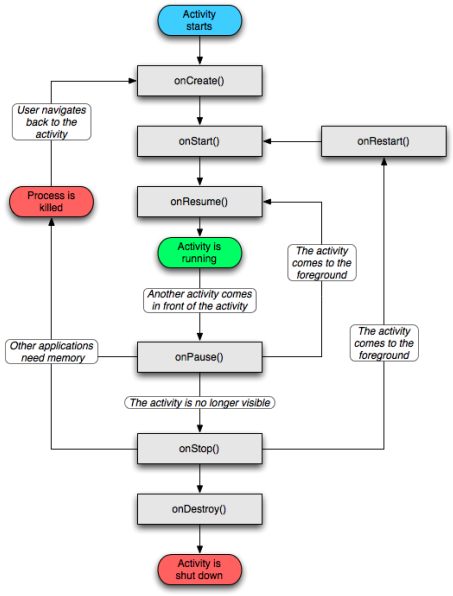
III. Software Framework for Android

This section describes the components that have been used for the proposed framework. First, we will give an overview on the Android life cycle and identify which of the life cycle callback methods are necessary for sensing user interaction. Second, we will go into more details on AOP and outline how this method can be used for transparently sensing interactions on mobile phones. Third, the component-based software architecture of our framework will be presented.

A. Understanding the Android Lifecycle

Android applications are written in Java and compiled into Android packages. Code within a single package (i.e. an *.apk file) makes up one application. When the application is downloaded from the market and installed on the device, it lives in its own security sandbox [8]. Android applications consist of various application components, of which activities and intents are relevant to the presented framework.

Figure 1. An Android activity consists of various lifecycle methods that are called if the view or page is created, displayed, sent to the background or destroyed. Source: [8]



1) Activities: An activity represents a single screen with a user interface. For instance, a ticketing application for cable cars might consist of one activity that shows a list of bought tickets, another activity to buy tickets, and a further activity for displaying the ticket details. Although activities can work together to provide a cohesive user experience, each one is independent of the others. As a consequence, if the right permissions are set, each activity can be launched from a different application [8]. The life cycle of an activity (cf. Fig. 1) can be managed by implementing callback methods. Basically, activities can exist in three different states [8]:

- **Resumed**. The activity is in the foreground and has the user focus. This state is referred to as "running".
- **Paused**. The activity is completely alive, its object is retained in memory, it maintains all state information and it remains attached to the window manager. However, it is covered by another activity that is in the foreground and has the focus, like on a stack. If the resumed activity is partially transparent or does not cover the whole screen, the paused activity is still visible.
- **Stopped**. The activity is in the "background", as it is completely obscured by another activity. In contrast to a paused activity, the stopped activity is not attached to the window manager. Although its object is retained in memory unless it is killed by the system in low-memory situations, it is no longer visible to the user.

Some of the life cycle callback methods (i.e. *onCreate()*, *onDestroy()*, *onResume()* and *onPause()*) are required for the presented framework, as they represent the points for source code injection.

2) Intents: Intents are asynchronous message calls responsible for activating activities [8]. Intents are used to bind individual components (i.e activities) to each other at runtime, regardless of the owner of the components. Thus, intents can also bind activities that do not belong to the application which created the intent. For instance, an application can create an intent that binds an activity that is owned by the application to another activity that is owned by the camera application.

Intents define messages to either activate a specific component or a specific type of component. Thus, it is possible to start activities explicitly by providing the activity object, or implicitly by simply providing the namespace as a string (e.g. "at.fhooe.MyActivity") that describes the location of the activity within the Android application package. To sum up, intents are used to connect activities to each other. Thus, the result is a navigational graph consisting of nodes represented by activities and edges represented by intents.

B. Use of Aspect-Oriented Programming

AOP allows to write independent software modules. In contrast to object-oriented programming (OOP), the program modules do not have to be referenced by components that make use of these modules (i.e. by declaring a class member variable, instantiating objects and calling object methods). These modules are called aspects [9]. Aspects are added to applications based on a so-called join point model (JPM). A join point is a well defined point in the program flow of an application. Thus, the JPM consists of the points where code from aspects is inserted into the application's code. Basically, the JPM comprises a list of methods, where

aspect code will be inserted. In other words, a JPM can be conceived as a list of line numbers where the aspect-oriented compiler adds code before the Java compiler produces the executable application package (cf. Fig. 2).

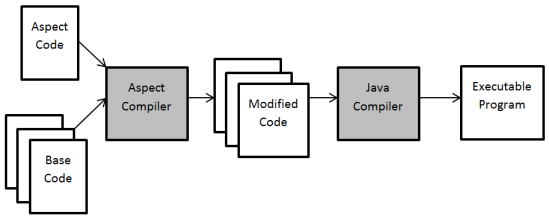


Figure 2. Aspect code is merged with application code at compile time, which is referred to as compile-time weaving. The aspect weaver produces code that can be compiled by the Java compiler.

In the case of Android applications, four point cuts that are crucial for the life cycle of Android applications have been identified. They are the life cycle methods *onCreate()*, *onResume()*, *onPause()* and *onDestroy()*. As described in Section III-A, *onCreate()* and *onDestroy()* indicate when an activity has been newly created or destroyed. As activities are loosely bound, these methods provide the only option to detect when an application has been launched or terminated. Therefore, a custom-developed application monitor has to count the number of created activities and reduce the counter if activities are being destroyed. If the counter equals zero, the application has been closed. The other two methods indicate which one of the activities is visible and has the focus to get feedback about user activities on the screen.

C. Framework Architecture

The architecture of the mobile framework is based on a modular concept (cf. Fig. 3) to handle data collection, life cycle injection and data transmission. The aspect layer on top of the framework is the only connection point between the framework and the host application. This layer is responsible for injecting code at the life cycle join points as described in Section III-B.

Application Layer	Android Host Applications				
Aspect Layer	Life Cycle Instrumentation				
Core	(An)Druid Kernel				
Android Toolbox	Credentials Service	IO Service	Activity Logger	Application Monitor	Network Service
Framework Toolbox	Name Service		Event Manager	Logging Service	
Implementation Layer	Service		Updateable Service	Console Logger	Android Logger

Figure 3. The mobile part of the proposed framework is based on a component-based architecture that consists of independent modules.

A central kernel handles the administration of different modules (i.e. services) that can be installed at the framework kernel at runtime. The framework toolbox contains multiple services that provide device functionality to services located in the Android toolbox (e.g. network access, logging functionality, file access, etc.). The android toolbox consist of various services used for interaction sensing such as monitoring users' navigational behaviour, the time they spent on screens or components they interact with.

D. Discussion

AOP covers concerns that are not well captured by traditional programming methodologies like OOP. For example, concerns like security cut across the natural units of modularity. Therefore, AOP is a way of modularizing cross-cutting concerns. For OOP, cross-cutting concerns are difficult to handle as they cannot be easily turned into classes. These concerns are mostly cut across classes, so they are not reusable and it is not possible to redefine them without major changes in all classes that make use of these concerns. Cross-cutting concerns spread throughout the program in an undisciplined way [10].

Where OOP modularizes common concerns in software design, AOP turns out to be a good approach for modularizing cross-cutting concerns. Compile-time weaving (i.e. adding the framework code at compile time) was chosen for the implementation of the presented framework, as it turned out to be the easier to implement than other approaches (i.e. post-compile weaving and load-time weaving). Moreover, post-compile weaving and load-time weaving do not offer additional advantages compared to compile-time weaving for our purpose, and load-time weaving is even not applicable for Android applications due to platform-specific limitations caused by the virtual machine on which Android applications run.

The architecture implemented for the mobile framework proves to be very scalable and extensible. New components can be implemented based on the presented service interfaces and installed at the framework without the need of changing the framework itself.

IV. Framework Evaluation

In order to get results concerning the feasibility and robustness of our framework, we tested it against various technical aspects including memory consumption, framework size and network payload. In order to provide measurable and comparable data, the framework has been added to the ScotDroid application, an Android ticketing application for buying public transport tickets in Austria[2].

This section provides details on the technical impact of the framework on the host application. This part of the section will outline the technical requirements that are presumed for the integration of the presented framework into an Android host application. The testing results are compared in terms

[2]https://market.android.com/details?id=at.fhooe.scotdroid

of static memory consumption (i.e. how much the size of ScotDroid increases by adding the framework), dynamic memory allocation (i.e. how much memory is allocated by the framework at runtime) and the network payload (i.e. how much network traffic is caused by the presented framework).

A. Evaluation Results

With regard to the static memory consumption, the original version of the ScotDroid application has a size of 1836 kB. Adding the framework source and the AspectJ runtime library, which is used for compilation only, leads to a size of 1968 kB. This means that the size of the modified version of ScotDroid has increased by only 132 kB, which is a growth of about 7%. Besides the allocation of these 132 kB, no further resources were allocated or required.

Version	Heap Size	Allocated	Free	%Used	#Objects
Original	5.422	3.370	2.052	62.15	64029
Original	5.871	3.645	2.226	62.09	66986
Original	5.734	3.621	2.113	63.15	68143
Original	6.035	4.004	2.031	66.35	75810
Original	6.242	4.088	2.154	65.49	78620
Modified	5.047	3.213	1.834	63.66	60420
Modified	5.344	3.606	1.738	67.47	64057
Modified	5.633	3.798	1.835	67.43	68273
Modified	5.949	4.111	1.838	69.11	70369
Modified	5.949	3.896	2.053	65.49	70470

Table I
MEMORY MEASUREMENTS ON AN HTC INCREDIBLE S WITH ANROID 2.3.3 FOR SCOTDROID, WITH AND WITHOUT THE FRAMEWORK.

For dynamic memory allocation, profiling the Dalvik Debug Management Service has been used to analyse the behaviour of the garbage collector and the heap. Therefore, multiple heap dumps have been created during the use of ScotDroid. Fig. 4 shows the memory allocation behaviour measured at intervals of five minutes during the use of the ScotDroid application on an HTC Incredible S mobile phone. The table shows the total size of the heap space in megabytes, the currently allocated amount of memory in megabytes, and the remaining amount of memory in megabytes before the heap size is reached. Moreover, this figure illustrates the number of currently instantiated objects.

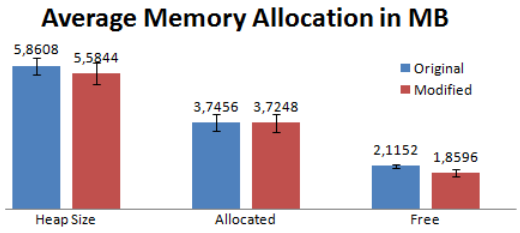


Figure 4. This figure illustrates the heapspace behaviour of ScotDroid with and without the proposed framework.

When the original version of ScotDroid is compared to the version that was modified with the framework, hardly any difference between the average memory allocation behaviour can be observed. As assumed, the modified version of ScotDroid uses a bit more space as more objects are allocated, for which reason a few kB less space remains free. However, the results show that adding the proposed framework to the ScotDroid application does not negatively affect the memory allocation for the process. Fig. 4 visualizes the average memory allocation from Table I, including the standard deviation computed on the measured data sets.

The size of transferred objects strongly depends on the kind of application that is monitored. In the case of ScotDroid, the average amount of data per application usage sent over the network is between 2 kB and 5 kB. More significant are the database statistics. 300 application users that used the application for one week caused 2543 database entries to be created. However, the size of these 2543 database entries is only 2 MB.

B. Discussion

First on-board tests showed that the framework requires very little memory. Especially for larger applications, the framework only requires a fractional amount of the space that the host application requires. In case of the ScotDroid application, the mobile framework enlarges the original application by just 7%.

Moreover, the proposed framework obviously does not affect the memory allocation at run-time. Heap sizes and allocated memory do not differ significantly when the proposed framework is added to the ScotDroid application. A slight difference can only be found for the average amount of allocated objects.

To sum up, the performance of the host application is not negatively affected or influenced by the framework. Also self-introduced exceptions did not affect the host application but only terminated the framework. Consequently, the framework can be considered to be stable and feasible from a technical point of view.

V. RELATED WORK

In the following, two closely related approaches are surveyed and compared to our solution.

In 2009, Balagtas-Fernandez et al. [11] presented an Android-based method and framework to simplify usability analysis on mobile devices. Their framework is a logging system that records usability metrics. Belagtas-Fernandez et al. pointed out that manually inserting source code into large applications is tedious and complex. Basically, our framework is also based on a logging system for sensing user interaction. In contrast, our approach uses AOP to reduce the complexity for adding source code into existing host applications. Hence, our framework can be added automatically, whereas the solution of Balagtas-Fernandez et al. requires developers to manually add framework source code to the application.

In 2007, Zduniak [4] presented a framework for automated GUI testing of Java ME applications. Zduniak developed a "record and replay" system that is able to inject source code into applications at byte code level. Hence, his framework can be added to *Java ME* applications after they are compiled or even at runtime. This approach is similar to AOP, because instead of injecting the life cycle at design time, Zduniak tried to extend Java ME classes by parsing and manipulating Java bytecode after host applications are built. However, this strategy cannot be applied to Android because for the *Dalvik VM*, which is used as virtual machine for Android, a scheme for bytecode generation is not provided. Thus, it is not possible to dynamically generate and load Java byte code at runtime into a running *Dalvik VM* [12]. Moreover, Zduniak's framework is limited to the recognition of user input of form-based Java applications, as he tries to get information about user input and not the users' operational behaviour. This means that with this framework, developers are capable of reconstructing user input for single users, but sensing of user interaction is not enabled. With our framework, we are able to reconstruct user input for single users as well, but in addition to that, we can also provide aggregated results based on the users' navigational behaviour which goes beyond form-based applications. Moreover, our approach can be applied to any platform that provides developers with AOP such as Windows Phone 7, iOS or Blackberry.

VI. Conclusions and Future Work

Based on the results from Section IV, we consider AOP as a valid approach for the creation of an independent framework that can be used for the automatic collection of usability data. As AOP does not negatively influence the host applications' performance, it might also be feasible for different platforms such as Windows Phone 7, the iOS or Blackberry as AOP frameworks are also available for them. In fact, we are currently porting the presented framework to Windows Phone 7, which requires an evaluation of the life cycle of Windows Phone 7 applications and the feasibility of AOP regarding this platform.

Additionally, first experiments showed that sensing user interaction at application granularity (e.g. navigation paths, navigation errors, etc.) is already a good starting point for tool-based usability evaluation. However, for more detailed information, user behaviour has to be captured at a finer level of granularity. Therefore, functionality for sensing user interaction at the granularity of activities is required and will be implemented (e.g. button clicks, text field focus, click positions, etc.) in the next stage of development.

Acknowledgements

The research presented is conducted within the Austrian project "AIR – Advanced Interface Research" funded by the Austrian Research Promotion Agency (FFG), the ZIT Center for Innovation and Technology and the province of Salzburg under contract number 825345.

References

[1] H. B.-L. Duh, G. C. B. Tan, and V. H.-H. Chen, "Usability evaluation for mobile device: A comparison of laboratory and field tests," in *Proceedings of the 8th Conference on Human-Computer Interaction with Mobile Devices and Services*, ser. MobileHCI '06. New York, NY, USA: ACM, 2006, pp. 181–186.

[2] G. McGlaun, F. Althoff, B. Schuller, and M. Lang, "A new technique for adjusting distraction moments in multitasking non-field usability tests," in *CHI '02 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '02. New York, NY, USA: ACM, 2002, pp. 666–667.

[3] A. Kaikkonen, T. Kallio, A. Kekäläinen, A. Kankainen, and M. Cankar, "Usability testing of mobile applications: A comparison between laboratory and field testing," *Journal of Usability Studies*, vol. 1, no. 1, pp. 4–16, November 2005.

[4] M. Zduniak, "Automated gui testing of mobile java applications," Master's thesis, Poznan University of Technology Faculty of Computer Science and Management, 2007.

[5] B. Morris, M. Bortenschlager, C. Luo, J. Lansdell, and M. Sommerville, *Introduction to Bada: A Developer's Guide*. John Wiley and Sons, 2010, ch. A Software Engineering Model for Mobile App Development, pp. 463 – 470.

[6] D. Madrigal and B. McClain, "Usability for mobile devices," September 2010, http://www.uxmatters.com/mt/archives/2010/09/usability-for-mobile-devices.php.

[7] Gartner, "Gartner says 428 million mobile communication devices sold worldwide in first quarter 2011, a 19 percent increase year-on-year," May 2011, http://www.gartner.com/it/page.jsp?id=1689814.

[8] Google, "Android application fundamentals," March 2011, http://developer.android.com/guide/topics/fundamentals.html.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, ser. ECOOP '97. London, UK: Springer-Verlag, 1997.

[10] Xerox, "The aspectj programming guide," Palo Alto Research Center, 2003, http://www.eclipse.org/aspectj/doc/released/progguide/index.html.

[11] F. Balagtas-Fernandez and H. Hussmann, "A methodology and framework to simplify usability analysis of mobile applications," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 520–524.

[12] J. Wilson, "Issue 6322: Api to generate dalvik bytecode at runtime," Android Enhancement Request, Jan 2010, http://code.google.com/p/android/issues/detail?id=6322.