OS Upgrade | Test Coverage | Black-box Test

Collaboration Projects

Efficient Application Testing for OS Upgrades

Application development accompanied by OS upgrades for smartphones requires the use of many test items to detect bugs that cannot be predicted solely on the basis of technical information released by the OS provider. This requirement drives up costs in application development, so to keep costs in check, we propose a method for extracting test items affected by OS code differences between the old and new versions of the OS before and after an upgrade. In this method, we first make an association between the application code affected by the upgrade and the target test process and then use test coverage information of affected application code for each test item. This proposal was developed and implemented in a tool through a joint-research partnership formed with Systems Engineering Consultants Co., Ltd. in January 2015.

Communication Device Development Department

Koichi Asano
Shinya Masuda
Mitsuhiro Ogata
Kazumasa Kobayashi

1. Introduction

In application development accompanied by OS*1 upgrades for smartphones, many test items must be used to deal with bugs that cannot be predicted solely on the basis of technical information released by the OS provider. This large number of test items has become a factor in increasing the cost of application development. In addition, the time period from the announcement of an OS upgrade to its market release tends to be short, so it has become very difficult to release an application supporting the post-upgrade OS (hereinafter referred to as "new OS") soon after the release of the new OS. Consequently, when working to keep up with OS upgrades, it is important that so-called upgrade development that deals with new functions provided by the new OS and changes to Application Programming Interface (API)*2 specifications be completed in a short time. This is essential to maintaining market competitiveness.

The production process of coding/compiling*3 in upgrade development involves editing work such as the addition of source code (hereinafter referred to as "code") to support new functions and the revision of code for existing functions affected by the upgrade. This work is performed based on technical information/materials [1] released by the OS provider and is followed by a testing process that may begin after compiling. At this time, actual execution of the application under the new OS may still

*1 **OS:** Software for managing an entire system by incorporating functions for basic management and control of a device and basic functions used in common by many software applications.
*2 **API:** A set of instructions, conventions, functions, etc. for use during programming.

uncover some bugs. One reason for this is that changes to operation specifications that actually exist may not be included in the technical information/materials. The fact is, totally unforeseen bugs may suddenly appear. Consequently, if the range of items targeted for testing cannot be specified, that range will inevitably broaden. That is, the number of test items subjected to a black-box test*4 tends to increase, which has been a factor in extending application development time and increasing development costs.

In this article, we focus on test items subjected to black-box tests and propose a method for specifying the range of testing and extracting test items. We then describe the implementation of a prototype system for Android™*5 applications to assess the usefulness of the method including its ability to reduce the number of test items. Finally, we present experimental results.

This proposal was developed and implemented in a tool through a joint-research partnership formed with Systems Engineering Consultants Co., Ltd. in January 2015.

2. Proposed Method

In this research, Step 1 obtains test coverage information [2] by associating the total application test table with the source code of the application (hereinafter referred to as "application code"). Next, Step 2 compares the application code with difference information between the old and new versions of OS code to extract all application code affected by the OS upgrade. Finally, Step 3 proposes a method for associating the extracted application code with the test process. Here, the test process may be joint/integration testing in application development or even acceptance testing performed on the side ordering the application development. The procedure for creating an application test table (extracted version) from the total application test table using the proposed method (steps 1 - 3 above) is shown in **Figure 1** and explained below.

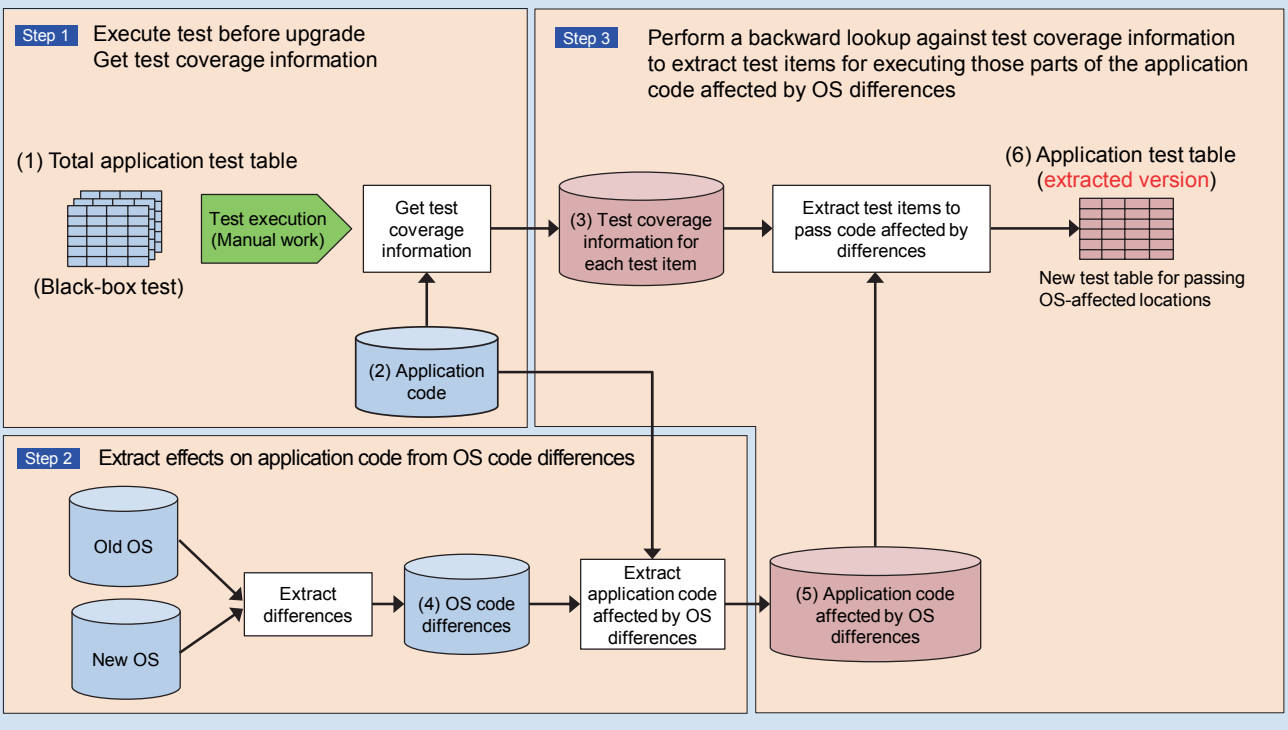

Figure 1 Extraction of test items affected by OS difference using test coverage information

*3 **Compiling:** The process of converting source code written in a programming language into an executable form after attaching a header, checking grammar, etc.

*4 **Black-box test:** An evaluation of a function seen as a unit from the outside without regard to its internal structure. Often used for joint testing, integration testing, and acceptance testing.

*5 **Android™:** A Linux-based open source platform developed by Google Inc. in the United States targeting mainly mobile information terminals. A trademark or registered trademark of Google Inc. in the United States.

2.1 Step 1: Get Test Coverage Information

Prior to the OS upgrade, the total application test table (Fig. 1 (1)) of the implemented test process is associated with the application code (Fig. 1 (2)). This work of associating the two is performed by the following procedure. First, when running the application to execute the test items in the total application test table, which lines of the application code are actually ran are recorded in units of line numbers. The content recorded here is called test coverage information for each test item (Fig. 1 (3)), which can be represented as shown in **Table 1**. It can be seen from this table that test item number 100 is appropriate when it is desired to run the 20th line of source code a.java®*6.

2.2 Step 2: Extract Effects of OS Differences

Extraction of differences between the old and new versions of the OS code can be represented as shown in **Table 2**. This is called OS code differences (Fig. 1 (4)). For example, it can be seen for OS code DEF.java that API internal processing changed after the upgrade, which means that a difference in operation may occur when called by the application. The effects of such OS differences on the application can be extracted by comparing OS code differences with the application code. Given application code as shown in **Table 3**, the application code affected by OS differences (Fig. 1 (5)) can be represented as shown in **Table 4**. It can be seen here that the OS upgrade affects the 20th line of application code a.java.

2.3 Step 3: Extract Test Items

The application test table (extracted version) (Fig. 1 (6)) can be extracted by comparing the application code affected by OS differences (Fig. 1 (5)) extracted in Step 2 with the test coverage information for each test item (Fig. 1 (3)) recorded in Step 1. **Table 5** is obtained from Table 1 and Table 4. It can be seen here that executing test item number 100 from among the test items

Table 1 Test coverage information for each test item (example)

Test item no.	Source code	Executed line number (test coverage information)
100	a.java	20
200	b.java	30

Table 2 OS code differences (example)

File name	Before change (Ver. 5.0)	After change (Ver. 6.0)	Difference
ABC.java	int ABC(a, b, c){	int ABC(a, b, c, d){	Add parameter
DEF.java	g = defexec();	g = def2exec();	Change internal processing

Table 3 Application code (example)

File name	Line no.	Source code statement
a.java	10	int r = 0;
	20	ret = DEF();
b.java	150	log(ABC(a, b, c));

Table 4 Application code affected by OS differences

File name	Line no.	Type	Description
a.java	20	Warning	Internal processing of called function DEF has changed
b.java	150	Fatal	Number of parameters of API ABC has increased

*6 **Java®:** An object-oriented programming language. Applications implemented in Java execute on a virtual machine, so they can operate on different platforms. Oracle and Java are registered trademarks of Oracle Corporation, its subsidiaries, and affiliates in the United States and other countries. Company and product names appearing in the text are trademarks or registered trademarks of each company.

in the total application test table is sufficient for testing the application code affected by the OS upgrade, and that test item number 200 need not be executed for this OS upgrade.

3. Implementation Method

3.1 Acquisition Environment for Test Coverage Information and Information Formatting

1) Acquisition Environment

Test coverage information is obtained using Java Code Coverage Library (JaCoCo)*7 [3] incorporated in Android Studio*8 [4]. This information can be obtained at test execution time through the Instrumentation Test*9 [5] in Android Studio, but this requires a continuous connection by Android Debug Bridge (ADB)*10 [6] between the PC used for operations during test execution and the Android terminal. As a result, power will still be supplied to the terminal when executing test items that require a low battery state thereby hindering the execution of some test items. To resolve this problem, we added a function to the Instrumentation Test that enables reset*11 and dump*12 operations against test coverage information whenever desired. In this way, we were able to avoid connection by ADB at test execution time. The constructed environment and actual procedure are shown in **Figure 2**. Before executing the test procedure, the tester performs a reset to delete test coverage information collected by JaCoCo. Then, during execution of the test procedure, test coverage information is collected by JaCoCo in memory and dumped to storage later.

Table 5 Application test table (extracted version) (example)

Test item no.
100

2) Formatting of Acquired Information

Test coverage information is obtained as a list of application code executed when executing test items. This test coverage information for all test items is then merged using application-code line numbers as keys to obtain a

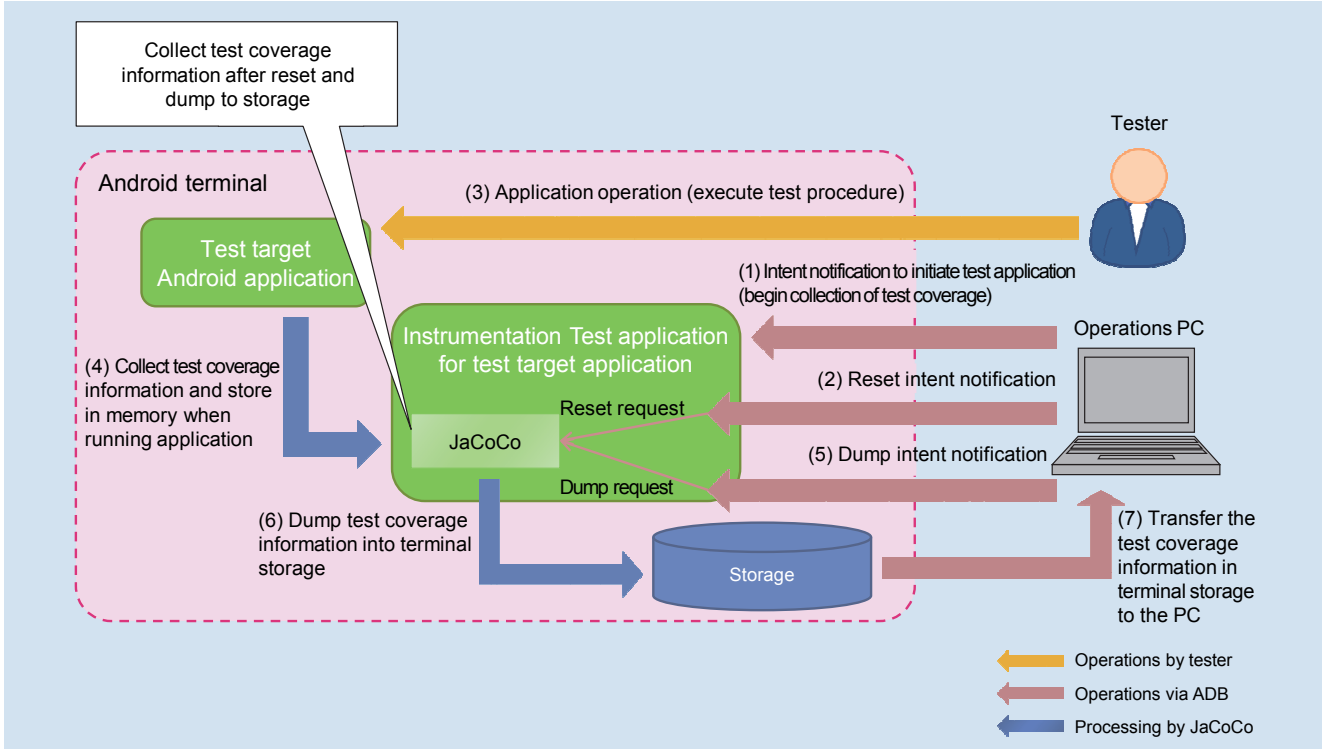

Figure 2 Environment and procedure for acquiring test coverage information at test execution

*7 **JaCoCo:** A library for obtaining test coverage of Java source code.
*8 **Android Studio:** An integrated development tool for Android applications.
*9 **Instrumentation Test:** A mechanism for performing automatic testing of Android applications.
*10 **ADB:** A tool included in the Android SDK capable of executing shell commands, performing file transfers, etc.
*11 **Reset:** An operation that deletes test coverage information saved in storage. Performed to prevent a mix-up with test coverage information recorded for other test items.
*12 **Dump:** An operation that saves test coverage information in storage. Used to record test coverage information for each test item.

list of application code passed at the time of test item execution (**Table 6**).

3.2 Procedures for Extracting OS Code Differences and Application Code Affected by Those Differences

Following the flow shown in **Figure 3**, OS code differences are extracted from OS source code before and after the OS upgrade as a list of classes*13 and methods*14 affected by classes and methods with differences.

Now, the portions of application code using OS code differences (classes and methods) are extracted to obtain a list of application code affected by OS code differences (**Table 7**).

3.3 Procedure for Extracting Test Items Affected by OS Code Differences

Finally, the lists obtained in Table 6 and Table 7 are merged with application code numbers as keys to extract

Table 6 List of application code to be passed at time of test item execution (example)

Application code		Test item no.1	Test item no.2	…	Test item no.X
Source file name	Line no.				
AAAAAA.java	10	○			○
	20	○	○		
	100		○		
BBBBBB.java	5		○		○
	15	○			○
	25	○			
⋮					

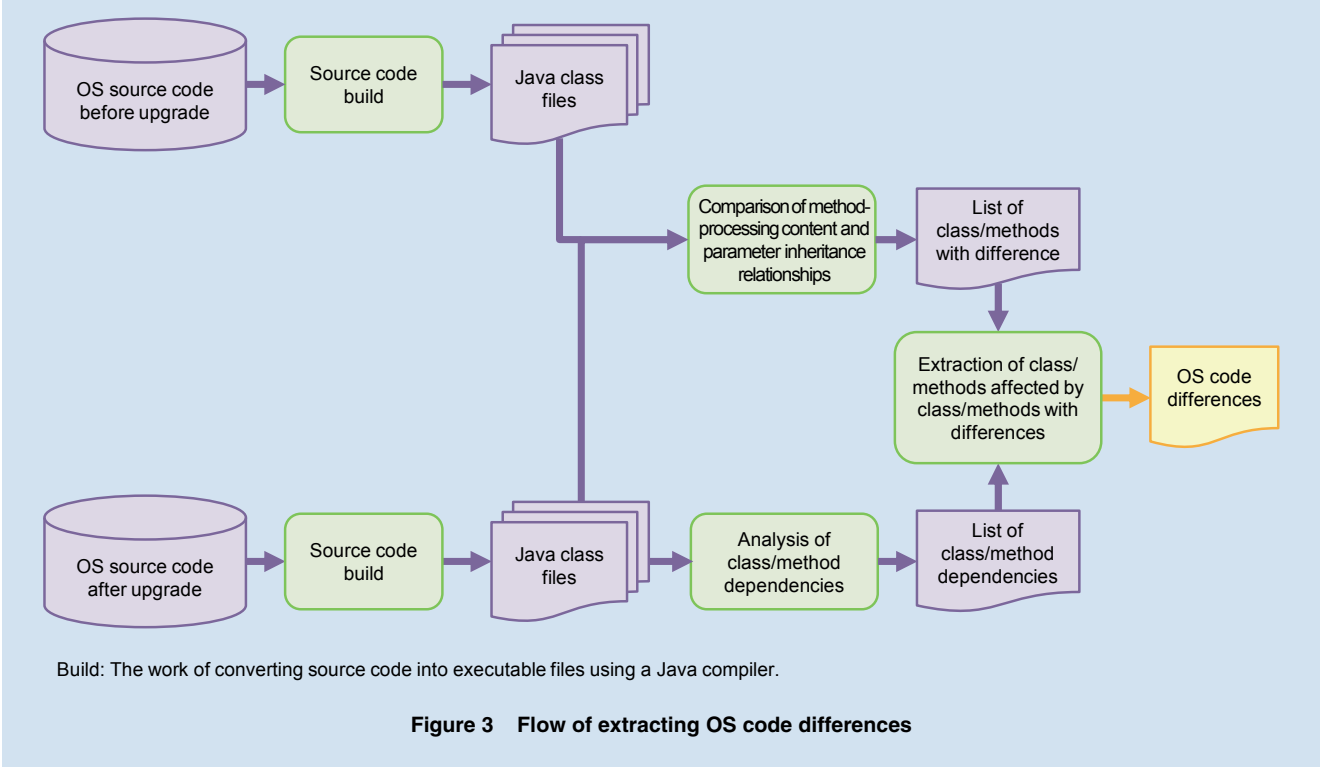

Figure 3 Flow of extracting OS code differences

*13 **Class:** Specified group of objects having similar states and behaviors in object-oriented programming.

*14 **Method:** Behavior of objects in object-oriented programming.

those test items affected by OS code differences (**Table 8**).

4. Experiment

We applied the procedure of this research to various Android applications at the time of the OS upgrade from 5.1.1 to 6.0.0 and measured whether the range of testing was specified, and if so, the extent to which the number of test items to be executed were decreased.

4.1 Experiment Results

The results of extracting application code and test items affected by OS code differences using the procedure of this research are listed in **Table 9**.

The test items extracted in this experiment indicate that extraction can be performed with equivalent accuracy as existing methods and that testing range can be specified by the proposed method without missing items that lead to the detection of bugs. On the other hand, results for two out of the four applications showed a reduction in number of test items of zero while the other two applications showed a reduction of 2 - 4 items (reduction effect of 2 - 4%). In short, we were not able to obtain the reduction effect in test range that we originally expected.

This can be explained by noting that source code lines for initialization processing, screen-related base class processing, etc. would be passed when executing any test item and that such lines are affected by the OS. As a result, nearly all test items came to be misjudged as "affected by OS differences."

4.2 Response to Issue

To resolve this issue, we assume that the following relationships exist between application code and test items.

- Assumption 1: For a line of application code passed when executing many test items, only the

Table 7 List of application code affected by OS code differences (example)

Application code		Level of OS-difference effect	Description
Source file name	Line no.		
AAAAAA.java	10	3	Class implementation change in API return value
	100	5	API parameter change
BBBBBB.java	15	4	API internal logic change
	50	3	Constructor logic change
	100	4	Addition of API throws specification
⋮			

Table 8 List of test items affected by OS code differences (example)

Application code		OS difference effect?	Test item no.1	Test item no.2	…	Test item no.X
Source file name	Line no.					
AAAAAA.java	10	Yes	○			○
	20		○	○		
	100	Yes		○		
BBBBBB.java	5			○		○
	15	Yes	○			○
	25		○			
⋮						

intended operation of that line when executing one of those test items need be verified.

- Assumption 2: For a line of application code passed only when executing a specific test item, the intended operation of that line when executing that test item must be verified.

An abbreviated example of a procedure based on the above assumptions for reducing the number of test items that must be executed is shown in **Table 10**. A red frame marks a line of application code executed by only one test item.

The results of applying this procedure to the results of the above experiment to reduce test items to only those for which execution is absolutely necessary are shown in **Table 11**. A red frame encloses the number of test items after this reduction process for each of the applications in the experiment.

Examining these results, it can be seen that the reduction rate for the "Schedule & Memo" application having a small number of test items overall is low. However, a reduction effect greater than 70% was obtained for each of the other three applications, which indicates that a sufficient effect was obtained in making testing at the time of an OS upgrade more efficient (by reducing the number of test items to be executed). Additionally, on comparing the results obtained by executing test items manually with results obtained by the proposed procedure, a test item for which bugs were discovered under manual execution was found to be absent in the set of test items after reduction. While

Table 9 Experiment results

Application name	No. of lines of code		No. of test items	
	Affected by OS differences	Total	Affected by OS differences	Total
Disaster Kit	1,676	7,964	28	28
Schedule & Memo	11,208	75,125	19	19
Hanashite Hon'yaku	8,548	41,844	165	169
Voice UI	1,025	4,323	47	49

Table 10 Overview of method for reducing test items (example)

Application code		Test item no.1	Test item no.2	Test item no.X
Source file name	Line no.			
AAAAAA.java	11	○		○
	12	○	○	
	13		○	
BBBBBB.java	21		○	○
	22	○		○
	23	○		
⋮				
Item reduction possible?		No Must be executed based on Assumption 2	No Must be executed based on Assumption 2	Yes Can be executed by another test item based on Assumption 1

Line of application code executed by only one test item

Line of application code executed by more than one test item

the effect on testing quality of a reduced number of test items based on assumptions 1 and 2 has not yet been studied, application of the proposed procedure to the above four Android applications showed that the number of test items to be executed could be reduced while maintaining testing quality.

4.3 Future Outlook

On extracting a list of test items affected by OS code differences (Table 8), it became clear that application code that had not been tested before by existing methods despite being affected by an OS upgrade could also be extracted. The results of extracting untested application code from the results obtained in the above experiment are listed in **Table 12**. A red frame encloses the number of lines of untested application code despite being affected by OS differences for each of the Android applications in this experiment. Application of this method can detect a deficiency of test items in other applications too.

For the OS upgrade targeted by this experiment, no defects were released into the market even though we reduced the number of test items using the proposed method. However, with an eye to future OS upgrades, we plan to go in a direction somewhat opposite to test-item reduction and to add testing that would check an application for any room for making improvements to software quality.

5. Conclusion

This article proposed a method for extracting test items for applications affected by an OS upgrade and showed that application testing became more efficient when validating application operation with a prototype system incorporating that method.

Going forward, we plan to use the proposed method on the release of new OSs and apply it to more applications with the aim of decreasing the number of test items and preventing omissions in creating new items. We also plan to study ways of implementing the method for correct operation in many cases including automatic testing and thereby broaden the scope of its use.

References

[1] Android Developers website. https://developer.android.com/index.html
[2] K. Yasuda: "Concept and Practice of

Table 11 Experiment results after item reduction

Application name	No. of test items			Reduction rate
	No. of items requiring execution	Affected by OS differences	Total	
Disaster Kit	18	126	126	85.7 %
Schedule & Memo	16	19	19	15.8 %
Hanashite Hon'yaku	45	165	169	73.4 %
Voice UI	6	47	49	87.8 %

Table 12 Number of untested lines of application code affected by OS upgrade

Application name	No. of lines of code		
	Affected by OS differences		Total
	Untested	Tested	
Disaster Kit	790	886	7,964
Schedule & Memo	8,075	3,133	75,125
Hanashite Hon'yaku	5,503	3,045	41,844
Voice UI	0	1,025	4,323

Software Quality Assurance—Systematic Approach toward the Open Era—," JUSE Press, Ltd., 1995 (in Japanese).
[3] EclEmma: "JaCoCo Java Code Coverage Library." http://www.eclemma.org/jacoco/
[4] Android Studio: "Android Studio Overview." https://developer.android.com/studio/intro/index.html
[5] Android Studio: "Test Your App." https://developer.android.com/studio/test/index.html
[6] Android Studio: "Android Debug Bridge." https://developer.android.com/studio/command-line/adb.html