Automatic assessment of assignments for Android application programming courses

Matej Madeja, Jaroslav Porubän
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia

***Abstract*—This paper presents a solution of creating a testing environment for Android applications in programming courses. Appropriate testing methods and suggestions for the testing environment are consulted by the authors with a mobile application development company. The paper also analyzes basic student mistakes, looks for solutions to their detection by automated testing, and suggests appropriate test tools on their basis. At the same time, the paper contains an assessment and testing experience for particular tools. In addition, it compares the performance of emulator and real device tests, and the proposed tools are partially tested in particular course.**

I. Introduction

Mary Meeker's report [1] from May 2017 shows that on average American adult spent 3.1 hours per day on a mobile device in 2016 and this number is still increasing. The report also points to the fact that since 2011 Android is the most used mobile operating system and from April 2017 has the major market share among all operating systems in the world [2]. Due to the above popularity a research [3] with 400+ IT professionals has been conducted and 76% of them stated they develop native Android applications and companies in which they are employed produce mostly applications for Android platform. This is the reason why MOOC (massive open online course) are so popular in this business (see [4], [5], [6]).

Many IT companies in the area of our university demand experienced programmers [7]. Responding to this situation, our university launched the course *Application Development for Smart Devices* in 2015 with a major focus on the Android platform. One of the inconvenient tasks in programming courses is to evaluate the students' assignments. In the standard procedure, the teacher must run each program, enter test data and evaluate the correctness of the solution. As the course is usually attended by tens or hundreds of students, therefore, the evaluation of the assignments is very time-consuming.

Until today 2 runs of the course were conducted, where the assignments were evaluated in a typical way - by the teacher. It was clear that it is not possible to evaluate the students' solutions properly and objectively. There is no way to test all possible inputs, so the question arises: How to test such applications? Furthermore, when assignment is submitted by a student, it is often seen that the student did not do the job by himself. In the case of typical evaluation of assignment is this fact really difficult to prove. Automated testing can help us solve the problem of student's assignment originality through an automatic plagiarism detection (more in [8] and [9]). Test-driven development (students create implementation according to tests) also shows that it is beneficial for the quality of programs [10], so automatic student's solution testing should also improve their quality of program functionality (probably source code, too).

Related research, with focus on related surveys of mobile application testing and teaching, is presented in Section II. Later in Section III we briefly describe an application for which testing tools will be suggested. In Section IV we analyse the most common mistakes of students, Section V is dedicated to the design of the testing environment and partial results are described in Section VI. Expectations and plans for the future are discussed in Section VII and conclusions in Section VIII.

II. Related work

Testing mobile applications is common, but not with a focus on automatic assessment of assignments in programming courses. We first searched for existing automated testing solutions for students' assignments in MOOC for Android programming, but none of the courses found provide assessment of student assignments by automated testing. These courses only deal with application development itself and mostly rely on manual app testing (either by teachers or directly by student in his or her own interest). Although some courses contain automated tests development in the curriculum (e.g. [11], [12], [13]), but again, none of the courses validate students' solutions by automated system.

Unfortunately, we did not find any research that addresses the same issue. We also sought inspiration at prestigious universities such as Stanford University, MIT[1], or Harvard University, but there was nothing to suggest from the general information that they use automated tests in similar courses and we have no access to their internal system. Nevertheless, there are a number of practical tools that could be explored in terms of target use.

Evidence that the topic of testing (of mobile apps) is generally up to date is that many developers write about it on their blogs and business pages. Already in 2007, Pecinovský [14] claimed that students should be taught by the step-by-step

[1]Massachusetts Institute of Technology

978-1-5386-0889-0/17/$31.00

coding method. Ten years ago, in 2017 DZone team issued a guide called *Automated Testing* [15] in which they claim that there is no longer any other way to develop software only through Continuous Delivery or Integration. Automated testing is essential to DevOps and Continuous Delivery and in order to integrate continuous testing effectively into a DevOps toolchain, the following essential features are key to evaluating an automated testing platform:

- Support for a variety of languages, tools, and frameworks.
- Cloud testing.
- The ability to scale rapidly.
- Highly automated.
- Security.

Of the over 400 IT professionals questioned in DZone's research, most said they generally use integration, component or performance testing in Continuous Delivery. Furthermore, Sauce Labs issued ebook *Mobile app testing: Main challenges, different approaches, one solution* [16] in 2017 in which it mentions that the biggest challenges for automated testing applications are:

- time saving,
- cost reduction,
- repeatability of tests,
- increased coverage of app features,
- re-usability.

Nevertheless, they (and other researches, e.g. [17], [18]) claim that manual testing for mobile apps is still unnecessary for some device-specifc function, such as location data or other environmental sensor data. The ebook also provides an overview of testing tools and solutions for testing infrastructure. Since Android is the most popular mobile OS (2011)[2], many other researches for automated testing has been done, such as [19], [20], [21].

In 2016 Wilcox described in paper *Testing Strategies for Automated Grading of Student Programs* [22] a set of strategies for testing students' programs that are more effective than regression testing at providing detailed and relevant feedback to students. In addition, he discussed some of the issues that arise in the context of automated grading and their solutions, such as grading performance, non-terminating programs and security issues. In recent years there are many other papers, e.g. [23], [24], [25], in which authors look at the automatic assessment of assignments from different perspectives.

III. Application Makacs

In the first two runs of the course *Application Development for Smart Devices* (2015, 2016), the main task was to create a sports monitoring and tracking application called *Makacs*. The implementation of this application was designed to bring the student into many topics in the development of Android applications. The application had the following requirements:

- min. 5 activities, of which at least 1 will include calorie counting,
- min. 1 service (recommended for counters - duration, pace, distance, calories),
- min. 1 broadcast receiver,
- min. 1 activity with own list implementation,
- data persistence with *SQLite*[2],
- use of min. 1 sensor,
- communication with an external web service through the API,
- min. 2 languages variants,
- min. 1 custom extension.

The app have to be targeted for Android devices with API 19+ and did not have any other implementation restrictions, it only had to meet the requirements and features had to be realistically usable. The creativity and the uniqueness of the implementation were left to the student, while the originality of the solution was assessed, of course, by teacher's subjective opinion. Because we have decided to find a solution for automatic assessment of students' assignments students will have to be restricted to a greater extent (due to the testing environment), which is a risk that students will be less creative.

IV. Most common students' mistakes

Based on the Section II, we conclude that there are many solutions and insights into test systems (or environments) in the learning process, but none of mentioned is used to test mobile applications. Our first question, then, was: What to test in student solutions? In generally the most common Android app failures are the following [26]:

1) Application failure when installing.
2) Application crash during execution.
3) Problems with scaling or deploying elements on the screen.
4) Non-responding application from unavailable sources.
5) Problems of viewing content in landscape or portrait mode.

These were all common issues for apps that people hate the most. Practical solutions in Section II show mobile applications are tested, but the testing of students' programs may lead to unique situations which, in practice, do not need to be treated to such an extent. Due to the possibility of occurrence such unique situations, we have been consulting the most often mistakes of new Android app developers with mobile development IT company *Wirecard*. After collecting the recommendations, we first selected 4 random student apps from the course in 2015 and 2016 and tested them manually. Selected applications have encountered issues such as navigation problems, data loss on activity restart (device rotation or change of system language), and some logical issues (such as device uptime dependency for counting duration of sport activity).

Since app errors were approximately distributed in the same number, we decided to test another 4 apps to get more accurate results. After trying them some new issues appeared such as the inability to install the app and forgotten service at activity restart, caused by change of system language. In generally, other issues were only repeated. Because we wanted to make

[2]https://www.sqlite.org/

TABLE I
MOST COMMON STUDENT'S ISSUES IN MOBILE DEVELOPMENT.

Issue	Failed Applications
Navigation & UX	7
Data loss on activity restart (e.g. device rotation)	6
Logical problems (e.g. usage of device variable data)	3
Unable to start or install app	2
Forgotten services at activity restart	1

sure we will not find other issues yet, we tried to test two more apps where there was no new issue found. Together, 10 applications were tested and the test results are shown in the Table I.

Manual testing was carried out on 2 facilities:

1) Nexus 5, API 25, Android 7.1.1, emulator.
2) Prestigio 5453 DUO, API 19, Android 4.4.2, real device.

Based on these results, we would like to concentrate mainly on these issues in test cases for our automated testing environment.

V. TESTING ENVIRONMENT

Now we know the most common mistakes of students and applications and the next question arise: How we can build a test environment to detect these issues/errors? The differences between testing of desktop programs and mobile applications, as well as problems arising from the testing of mobile applications, are described in more detail in Wasserman's article *Software engineering issues for mobile application development* [27]. Some of these issues and new problems found by us are discussed below.

A. Static testing

The plan for the course curriculum is to have 4 deadlines to submit partial assignment solution during the semester, which will be tested in testing environment. At the beginning of the course it is a big problem to motivate students to work. Lectures compared to seminars are usually not at the same time level, meaning that students often lack the theoretical knowledge to start programming. However, when designing mobile applications, students can design a UI for their application and become familiar with UI components in the IDE, where the work rests solely on editing XML documents.

We have devised the task at the beginning seminars of the course, which will be able to test the testing environment. We have created a task where the students have to create their own application design and prepare it in IDE. Created (or generated) XML files can then be uploaded and the test platform can check them. The seminar documents describe exactly what elements must contain specific activity resource file (file defining UI elements) and what identifiers must be defined in it. The test system then tests only the uniqueness of the identifiers for every activity file and whether the identifier is associated with the correct element. This automated test layer (we call it *xmlchecker*) has a static character (static tests) and requires an exact directory structure with resource files. On the other hand, in order to check whether the students actually created the real-world design of the application, we require that screens of activities be sent and then the teacher check them manually (so that identifiers can not be fake-generated and sent for evaluation). This type of manual checking is not very time-consuming and we have judged it to be sufficient.

B. Test pyramid and tools selection

The other 3 submissions contain, in addition to XML checks, real test of the functionality of the students' source code, so in the following testing we will use several tools. The question arises how the test structure should look like. Typically, test cases are divided into several layers. For Android apps, the following test layers are common [26]:

- unit tests,
- integration tests,
- operational tests (also called functional or acceptation),
- system tests.

In our case, we chose to use the concept of the *test pyramid* [28]. The concept includes the following 3 tiers:

- unit tests,
- integration tests,
- UI tests (or system tests).

When testing student assignments we do not need a separate layer for operational tests, because they will be included in UI tests (according to existing testing tools). The test pyramid concept says any automated testing strategy should have more low-level unit tests than high-level UI tests. From a viewpoint of the complexity of performing tests, unit tests are the easiest and fastest, on the contrary, performing UI tests takes a relatively long time. By using these tiers in our test platform, where hundreds of applications are tested and the idea of this concept is preserved, we can accelerate the testing process and increase test reliability (by testing the same or similar functionality by various tools from various tiers).

For listed test tiers of test pyramid we compared the various open-source test tools so that we can test the most functionality of student applications automated. We also consulted with *Wirecard* company. For unit testing, Google recommend to use *JUnit* with *Mockito* [29] tool to create mock objects (needed for tests where application context is needed). The purpose of unit test is to isolate the smallest source code units and check, even in isolation, they work correctly. In Android applications are often even the smallest tasks depended on the context of the app, in which case *Mockito* can help us.

For integration tests, *Wirecard* advised us to use *Robolectric*[3] framework with whom they had a wealth of experience. Due to the close offer of similar tools, we have not found a suitable competitor, so we have decided to use this tool.

As for the last layer of UX testing, choosing a test tool was not as simple as the previous ones. Because there is a lot of UI testing tools and they overtake each other in their features, we have decided to do our own comparison [30]

[3]http://robolectric.org/

of these tools in 2017. We have looked at this tools from a practical point of view, and in particular from the point of view of their capability to test. The most popular tools were *Appium*[4], *Robotium*[5] and *Espresso*[6]. Though, the most important indicator for us was their ability to cover the widest possible range of different test cases. We looked at factors like app support type (native, hybrid), context (device, application), IDE intergration, support of OS vendor, emulator vs. real device testing, etc. The best tools from this point of view were *Espresso*, *Appium* and *Robotium*. *Espresso* has the best overall score, so we decided to use it in our solution. It has the ability to record test cases, which greatly facilitates the creation of tests. In the UI tests, we also decided to use the *Monkey*[7] tool to perform stress tests (different and fast gestures above the app UI).

C. Writing tests

Because the proposal was not yet implemented in the course, we created a sample *Makacs* application that we experimented with. The goal of the test pyramid is appropriate for the assessment of student's assignments because unit tests could be done very quickly. First we started writing unit tests to get them as much as possible. In the sample app proposal, a method of calculating calories, change the units of distance (km, mi), weight (kg, lb), etc., was precisely defined. Nevertheless, after a short while, we found out that we were actually tested everything possible. Together, 6 different unit tests were made with a size of 5-16 lines. And there was a problem because the notion that UI tests would be even smaller was unrealistic. Nevertheless, we continued the chosen concept and tools.

Some tests need to be developed in conjunction with the *Android SDK*, but we do not want to deploy the application to virtual or real device, most often because it is time-consuming. In that case, we used the *Robolectric* framework, as a part of integration tests, that mocks the *Android SDK* and thus eliminates the `RuntimeException` exceptions resulting from the empty, so-called "stub" implementation of methods. Tests are running directly in JVM, so we did not need a virtual or real device yet. This tool is actually a headless UI test framework (possible to call action on UI element). When writing tests, we found that UI headless testing errors were detected, pointing to some problems in the test framework. We found out that problems occur with every new version of the *Android SDK* when a new implementation of stub methods is needed (mocking of *Android SDK*). Therefore, it is not a full replacement for UI testing tools because UI testing tools do not have such dependencies. The best choice is a combination of both approaches (headless and classic UI testing). However, this headless UI test has not been reliable due to the various UI elements in the various Android APIs and the resulting conflicts.

A great feature of *Robotium* was ability to automatically clear the test environment for each test (such as app settings). Problems, however, occurred when running multiple database access across tests because tests fell due to singleton implementation of the `DatabaseOpenHelper` class (due to usage *ORMLite*[8]). After each test performed, it is necessary to release the singleton manually.

At the same time, question arose how to test intents of individual activities or starting/stopping app services. From the student application, the desired action (new activity, system application request, etc.) is expected to be executed after clicking on a specific button (element ID). *Robotium* provides a great and simple solution for tracking intentions. Likewise, for services, it provides a stack of all existing services in the application. Implementation is more complicated during testing, because it is always necessary to erase the stack to test a particular service. Together, 11 tests for the *Robotium* tool were performed in a range of 2-10 rows. All unit and integration tests are able to run using the *gradle*[9] tool.

As part of the integration tests, we knew that after a specific action the service with that name started or stopped. However, we was not able to test the functionality of the service, which was a major part of the *Makacs* logic (calorie counting, distance, etc.). Despite the fact that *Robolectic* allow testing of service functionality, the solution was hard to implement. We found the *AndroidJUnit4*[10] tool, which is much easier to implement using a binder, on the other hand this tool needs an Android device (virtual or real). Together we created 6 tests for the main service of the 24-44 line size.

As the last and highest layer of test environment was UI testing. *Espresso* is a young test framework, which is also felt by the developer because the writing of the tests is very intuitive. Even general tests (such as navigation) can be created using a test recorder, which greatly accelerates the creation of tests. The only downside of recording is that it does not look for component identifiers but for their texts and order in a particular activity. This approach is not accurate, so you need to set the identifiers manually. However, it's faster than writing all tests manually.

The problem occurred when running multiple tests at once. *Espresso* remembers the status of the application's database as tests run on a specific device. This is undesirable in some cases (especially when modeling specific use case, e.g. data recovery from database of crashed app), so it is necessary to delete the database manually in these cases. Tests are triggered at random, so it is necessary to individually check before each test whether the application is in the required state to start particular test.

As part of the UI testing, we used the stress testing tool *Monkey*, which runs random 10,000 random gestures over each student solution to get the application into an unexpected error.

[4]http://appium.io/
[5]https://robotium.com/
[6]https://developer.android.com/training/testing/espresso/
[7]https://developer.android.com/studio/test/monkey.html
[8]http://ormlite.com/
[9]https://gradle.org/
[10]https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html

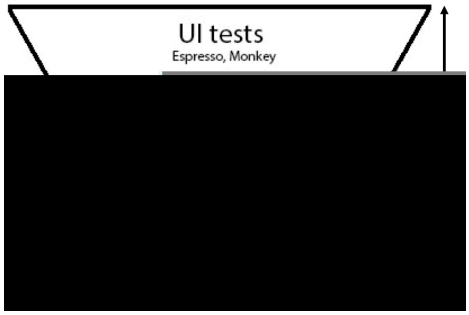

Fig. 1. Tiers of proposed testing environment.

Figure 1 shows the actual proposed structure of the test environment. As we can see, the testing enviroment in terms of the number of tests is the opposite of the test pyramid concept. This is due to the fact that if we do not want to limit the students' solution to a large extent in the assessment of students' assignments, we need to focus on testing that is not dependent on implementation details (at most IDs of UI components). We have also added static tests to the test process for tasks that a student can accomplish without the need of programming on a given platform. This result does not say that the concept of the test pyramid is poorly designed, but for the testing of students' assignments, in our case, we had a different character from the view of tests number.

VI. Partial experimental results

All these suggestions were prepared for the fall 2017 semester of the course. In order to know the assumption that the testing environment is capable of evaluating assignments, we performed a number of possible tests on a random sample. Some results of manual testing have been already described in Section IV, because based on these results, we focused on selecting test tools and implementing particular test cases.

From the proposed test tools, student solutions from 2015 and 2016 could be tested only by stress testing by the *Monkey* tool, which is not dependent on the specific implementation of the solution. We tested 10 randomly selected applications that were approved by manual teacher testing, with 50% of testing applications failing. In the test 10,000 random gestures were used over the application, and the most common failures were `NullPointerException`, `RuntimeException` and `SQLException`. These results indicate that *Monkey* has demonstrated the imperfection of manual testing and greatly enhances the objectivity of evaluating assignments.

In the typical testing process (companies), a few, mostly tens of applications are tested. However, in our case, we will test hundreds of the same applications several times a day, which can be time-consuming to perform. That's why we've run tests on the sample *Makacs* app to find out how long it takes to perform all the tests.

We've run tests on 3 real devices and 3 virtual devices (real device copies) where we tested sample app for 71 created tests. On virtual devices, we used Intel x86 Android system images, because of host's processor had mentioned instruction set. During the 10 measurements (Table II), we found that testing on a real device is approximately 2-3 times faster. Therefore, in addition to virtual devices (to perform tests on multiple devices), we will use the real device in particular. With 150 submitted solutions (expected number of students in 2017), our system can check on real devices on average for 6 hours 27 min 30 sec. It's quite long time, so we'll run tests probably every 12 hours.

TABLE II
Execution speed of 71 tests on different devices in 10 measurements.

Device	DT	Best Time	Avg. Time	Worst Time
Nexus 5, API 25,	VD	5 min 53 s	6 min 32 s	7 min 2 s
Android 7.1.1	RD	1 min 58 s	2 min 30 s	2 min 54 s
Prestigio 5453 DUO,	VD	2 min 12 s	2 min 52 s	4 min 1 s
API 19, Android 4.4	RD	2 min 23 s	2 min 59 s	3 min 30 s
Xiaomi Redmi 3s,	VD	4 min 12 s	5 min 48 s	6 min 8 s
API 23, Android 6.0	RD	1 min 52 s	2 min 16 s	2 min 36 s

DT - device type, VD - virtual device, RD - real device

During these tests, we encountered a problem with an unexpected throwing of exception that there is no activity opened. At the time of designing and testing the testing environment, *Espresso* developers were unable to resolve this problem, a few weeks later they fixed the bug. However, this issue still persisted on some devices. We explored that the problem occours while animations on a device are enabled, so it is necessary to turn off all device animations before running tests.

Other tests could not be tested on student solutions from previous years because the structure of the projects (classes, methods, IDs etc.) was not adapted to the proposed test environment.

VII. Future work

Improving the test process is an endless loop that can not be stopped. Based on our partial results, we will monitor students' responses to the test platform and customize test cases to bring maximum benefit to the student and teacher. The results from the real use in the course will be presented in the next paper where we expect feedback from approximately 150 students.

Given the problems with UI test speeds and their occasional unreliability with animations turned on (*Espresso*), we would like to compare and research different UI test frameworks in the future. Moreover, it might be interesting to require test cases by students in their assignments, than we could check these tests and use them to test other student assignments. Students will help us build tests and reveal deficiencies in testing process. About the evaluation of the tests made by the students is written by Smith et al. in paper [31] from 2017.

VIII. Conclusion

This experimental investigation has analyzed the most common mistakes of students who have become involved in previous runs of *Application Development for Smart Devices* course. Based on these mistakes and with the inspiration of

the *test pyramid* concept, which seemed appropriate for use in the given course, we designed a test environment and used the appropriate test tools for the proposed tiers. In the end, we have found that the number of tests in our platform is not identical to the pyramid's ideology.

Of all the suggested tools, it was possible to perform only *Monkey* stress tests on solutions from past runs of the course. The results of the stress testing of a small sample proved to be appropriate and pointed to the inaccuracy of manual evaluation of the assignments. At the same time, we looked at the length of testing on real and virtual devices, and it was definitely more reliable and faster to perform tests on a real device. The authors also evaluated the advantages, disadvantages and experience with testing tools.

ACKNOWLEDGMENT

The authors would like to thank all the participants in the research, especially students and teachers of *Application Development for Smart Devices* course. At the same time we thank *Wirecard* company for sharing their research results, experiences with mobile testing and any other methodical help. This work was supported by project KEGA 047TUKE-4/2016 Integrating software processes into the teaching of programming.

REFERENCES

[1] M. Meeker, "Internet trends 2017," CODE CONFERENCE, Tech. Rep., 05 2017.

[2] StatCounter, "Operating system market share worldwide," 09 2017. [Online]. Available: http://gs.statcounter.com/os-market-share#monthly-201101-201708

[3] J. Esposito *et al.*, *Mobile Application Development*, C. Candelmo *et al.*, Eds. DZone, 2016, vol. 3.

[4] M. Cook, "The 50 most popular moocs of all time," 04 2015. [Online]. Available: http://www.onlinecoursereport.com/the-50-most-popular-moocs-of-all-time/

[5] Coursera Inc., "Courses and specializations," 2017. [Online]. Available: https://www.coursera.org/courses?_facet_changed_=true&domains=computer-science&languages=en&query=android

[6] edX Inc., "Android courses search," 2017. [Online]. Available: https://www.edx.org/course?search_query=android

[7] M. Madeja, "Innovative approaches in introductory programming courses," Master's thesis, Technical university of Košice, 05 2015.

[8] C. Domin, H. Pohl, and M. Krause, "Improving plagiarism detection in coding assignments by dynamic removal of common ground," in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '16. New York, NY, USA: ACM, 2016, pp. 1173–1179. [Online]. Available: http://doi.acm.org/10.1145/2851581.2892512

[9] S. Mann and Z. Frew, "Similarity and originality in code: Plagiarism and normal variation in student assignments," in *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ser. ACE '06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 143–150. [Online]. Available: http://dl.acm.org/citation.cfm?id=1151869.1151888

[10] H. Munir, K. Wnuk, K. Petersen, and M. Moayyed, "An experimental evaluation of test driven development vs. test-last development with industry professionals," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '14. New York, NY, USA: ACM, 2014, pp. 50:1–50:10. [Online]. Available: http://doi.acm.org/10.1145/2601248.2601267

[11] L. V. Soham Mondal, "Android app development," 2017. [Online]. Available: https://www.springboard.com/learning-paths/android/

[12] D. C. Schmidt *et al.*, "Launch your android app development career," 2017. [Online]. Available: https://www.coursera.org/specializations/android-app-development

[13] F. A. Adrián Catalán, Noe Branagan, "Professional android app development," 2017. [Online]. Available: https://www.edx.org/course/professional-android-app-development-galileox-caad003x

[14] R. Pecinovský, "Zadávání a vyhodnocování úkolů při výuce oop," in *Počítač ve škole 2007*, Amaio Technologies, Inc. Nové Město na Moravě, Czech rep.: Počítač ve škole, 2007, pp. 1–4.

[15] J. Sugrue *et al.*, *Automated Testing*. DZone, 2017, vol. 1.

[16] Souce Labs, "Mobile app testing: Main challenges, different approaches, one solution," DZone, 09 2017.

[17] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 83–93. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771786

[18] B. Amen, S. Mahmood, and J. Lu, "Mobile application testing matrix and challenges," in *Computer Science & Information Technology*, vol. 5, 04 2015.

[19] M. Akourm, B. Falah, A. A. Al-Zyoud, S. Bouriat, and K. Alemerien, "Mobile software testing: Thoughts, strategies, challenges, and experimental study," in *(IJACSA) International Journal of Advanced Computer Science and Applications(ijacsa)*, vol. 7. SAI, 02 2016. [Online]. Available: http://thesai.org/Publications/ViewPaper?Volume=7&Issue=6&Code=ijacsa&SerialNo=2

[20] Y. Wang and Y. Alshboul. (2016, 02) Mobile security testing approaches and challenges. Gainesville, Florida, USA. [Online]. Available: https://www.researchgate.net/publication/277132880_Mobile_Security_Testing_Approaches_and_Challenges

[21] H. Muccini, A. D. Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *2012 7th International Workshop on Automation of Software Test (AST)*, June 2012, pp. 29–35.

[22] C. Wilcox, "Testing strategies for the automated grading of student programs," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: ACM, 2016, pp. 437–442. [Online]. Available: http://doi.acm.org/10.1145/2839509.2844616

[23] T. Rajala, E. Kaila, R. Lindén, E. Kurvinen, E. Lokkila, M.-J. Laakso, and T. Salakoski, "Automatically assessed electronic exams in programming courses," in *Proceedings of the Australasian Computer Science Week Multiconference*, ser. ACSW '16. New York, NY, USA: ACM, 2016, pp. 11:1–11:8. [Online]. Available: http://doi.acm.org/10.1145/2843043.2843062

[24] Y. Akahane, H. Kitaya, and U. Inoue, "Design and evaluation of automated scoring: Java programming assignments," *Int. J. Softw. Innov.*, vol. 3, no. 4, pp. 18–32, Oct. 2015. [Online]. Available: http://dx.doi.org/10.4018/IJSI.2015100102

[25] S. Gupta and S. K. Dubey, "Automatic assessment of programming assignment," in *ITCS, SIP, JSE-2012*, e. a. Natarajan Meghanathan, Ed. CS & IT, 2012, pp. 315–323.

[26] guru99.com. (2017, 08) Complete guide to android testing & automation. [Online]. Available: https://www.guru99.com/why-android-testing.html

[27] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 397–400. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882443

[28] C. Greb. (2016, 12) The 3 tiers of the android test pyramid. [Online]. Available: https://medium.com/android-testing-daily/the-3-tiers-of-the-android-test-pyramid-c1211b359acd

[29] Google Inc., *Test Your App*, 2017. [Online]. Available: https://developer.android.com/studio/test/index.html

[30] M. Madeja, "Testing of applications for os android," Master's thesis, Technical university of Košice, 04 2017.

[31] R. Smith, T. Tang, J. Warren, and S. Rixner, "An automated system for interactively learning software testing," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17. New York, NY, USA: ACM, 2017, pp. 98–103. [Online]. Available: http://doi.acm.org/10.1145/3059009.3059022