Research Article **Open Access**

Matej Madeja* and Jaroslav Porubän

Automated testing environment and assessment of assignments for Android MOOC

https://doi.org/10.1515/comp-2018-0007
Received February 23, 2018; accepted May 30, 2018

Abstract: This paper describes the design of a testing environment for massive assessment of assignments for Android application programming courses. Specific testing methods and tool suggestions are continuously consulted with *Wirecard* company, dedicated to the development of mobile applications. The paper also analyzes the most common mistakes of students and suggests ways to uncover them through tests. Based on these, it creates tests, compares the performance of the emulator and real device tests, and the proposed tools are partially retrospectively tested on assignments from the previous run of a particular Android application programming course. From partial results the paper suggests changes for the course in relation to the testing environment and deploys it in the background of the course alongside the manual evaluation. It describes testing experience, analyzes the results and suggests changes for the future.

Keywords: massive, android, application, testing, automation, assessment, MOOC

1 Introduction

Mary Meeker's report [1] from May 2017 shows that, on average, an 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 has been the most used mobile operating system and from April 2017 has the largest 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, 76% of whom stated they develop native Android applications and their employers mostly produce applications for the Android platform. This is the reason why MOOC (massive open online course) are so popular in this business (see [4–6]).

Many IT companies in the area of our university demand experienced programmers [7]. Addressing this need, 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 the evaluation of the assignments is very time-consuming.

In the first 2 runs 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. As there is no way to test all possible inputs, the question of how to test these applications arises. Furthermore, when an assignment is submitted by a student, it is often observed that the student has not completed the assignment by himself, which is quite difficult to prove in typical assignment evaluations. 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 an automatic plagiarism detection (more in [8] and [9]). Test-driven development (where students create implementation according to tests) also proves 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).

This research article is an extended version of a paper pubilshed in *2017 IEEE 14th International Scientific Conference on Informatics* [11] and extended with new experiences, adjustments and research results. Related research, with a focus on related surveys of mobile application testing, teaching and existing testing environments is presented in Section 2. Later in Section 3 we briefly describe an application and course specifics for which testing tools will be suggested together with the analysis of

***Corresponding Author: Matej Madeja:** Department of computers and Informatics, Technical University of Košice, 042 00, Košice, E-mail: matej.madeja@tuke.sk
Jaroslav Porubän: Department of computers and Informatics, Technical University of Košice, 042 00, Košice, E-mail: jaroslav.poruban@tuke.sk

most common students's mistakes. Section 4 is dedicated to the design and implementation, where we describe the process of writing tests and suggest changes to the course that could affect the test process. In Section 5, we describe the results from real deployment at the background of the course and we evaluate the results from testing experience. Conclusions, expectations and plans for the future are discussed in Section 6.

2 Related work

Testing mobile applications is common, but not with a focus on automatic assessment of assignments in programming courses. Firstly, we 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). Some courses contain automated tests development in the curriculum (e.g. [12–14]), but again, none of the courses validate students' solutions by automated system.

Unfortunately, we found no 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. Further, we have no access to their internal system where the information about the assessment of the assignments could be more detailed. Nevertheless, there are numerous practical tools that could be explored in terms of target use.

2.1 Testing Android apps in practice

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 [15] claimed that students should be taught by the step-by-step coding method. Ten years later, in 2017 DZone team issued a guide called *Automated Testing* [16] in which they claim that there is no longer any other way to develop software but 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 evaluate an automated testing platform:

- Support for a variety of languages, tools, and frameworks.
- Cloud testing.
- Rapid scalability.
- 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 an ebook titled: *Mobile app testing: Main challenges, different approaches, one solution* [17] 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, their research (and others', e.g. [18, 19]) claims that manual testing for mobile apps is still unnecessary for some device-specific 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 (since 2011)[2], many other researches for automated testing in business area have been done, such as [20–22].

2.2 Automated testing in programming courses

In their article, Gupta et al. [23] present an automatic assessment system for programming assignments, using a verification program with random inputs. The system consists of static and dynamic code analysis. In dynamic testing they provide random values as program input, and the generated input and output of the tested program is served as input for the verification program. They consider the verification program a sample solution for them, and this program verifies whether the student's solution gives the same result as the sample program that is considered reliable. In our environment, we use static and dynamic analysis, too. For example, user interface (UI) elements check could be subject to static analysis, whereas dynamic anal-

1 Massachusetts Institute of Technology

ysis could be included in unit tests. However, when testing Android apps, we will almost always need to mock context or some external services because most of the functionality is dependent on them. Our solution tries to cover this issues.

From a different point of view, classical unit tests require the implementation of specific method in tested solution from which a result will be expected after a method call. During Android app development most of the methods and classes are generated by an Integrated Development Environment (IDE) which is mostly related to the lifecycle of Android activity. The specific implementation may vary for different Android Application Programming Interfaces (APIs), so we need to focus more on functionality (black-box testing) than on calling specific methods (white-box) when testing these apps.

At our university, an automated testing environment has been developed in recent years, which is used for a variety of courses, eg. CS1, Java programming etc. In the testing environment, static and dynamic unit tests are performed, similar to those described by Gupta. However, the environment does not support integration testing or UI testing. For example, Java applications can perform UI tests even without a real screen because we can use a headless *xserver* in Linux. But real Android apps expect a real display. Our design looks for, compares and implements options to test the UI in the emulator and the real device. We cover integration testing, too.

In 2016 Wilcox described in *Testing Strategies for Automated Grading of Student Programs* [24] 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, nonterminating programs and security issues. In recent years there are many other papers, e.g. [25, 26], in which authors look at the automatic assessment of assignments from different perspectives.

3 Course specifics and Makacs application

The main task and assessment of the course *Application Development for Smart Devices* (2015 - 2017) is 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. In general the application had the following requirements in the first two runs of the course:

- 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 had 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 the 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.

3.1 Most common students' mistakes

Based on Section 2, we conclude that there are many solutions and insights into test systems (or environments) in the learning process, but none of those mentioned is used to test mobile applications. Our first question was: "What should we test in students' solutions?" In general, the most common Android app failures are the following [27]:

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 2 show mobile applications are tested, but the testing of students' programs

2 https://www.sqlite.org/

Table 1: Most common student's issues in mobile development.

Issue	Failed Applications
Navigation & user experience (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

may lead to unique situations which, in practice, do not need to be treated to such an extent. Due to the possibility of occurrence of such unique situations, we have been consulting the most frequent mistakes of junior 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. The 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 amount, 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 the change of system language. In general, other issues were only repeated. To ensure no further issues would be found, 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 Table 1.

Manual testing was carried out on 2 devices:

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.

4 Design and implementation

Having identified the most common student mistakes in the applications, the next question arose: How can we build a testing environment to detect these issues/errors? The differences between testing 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* [28]. Some of these issues and new ones found by us are discussed below.

4.1 Testing tiers and tools

Lewis [29] summarizes 40 kinds of tests in his book. Since there are many testing methods that test the same functionality from different viewpoint, we had to choose the right testing methods according to the most common mistakes of the students and separate these methods into tiers of the testing environment. Subsequently, we searched for the appropriate test tools for each tier.

4.1.1 Static testing

The plan for the course curriculum is to have 4 deadlines to submit partial assignment solutions during the semester, which will be tested in the testing environment. At the beginning of the course it is a big problem to motivate students to work. Seminars are sometimes not preceded by lectures, 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 divided the tasks at the beginning seminars of the course, which would be testable in our testing environment. We have created a task where the students have to create their own application design and prepare it in the IDE . Created (or generated) XML files can then be uploaded and the testing system can check them. The seminar documents describe exactly what elements must contain the specific activity resource file (file defining UI elements) and what identifiers must be defined in it. The testing 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 testing 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 teacher should 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.

4.1.2 Test pyramid and tools selection

The other 3 submissions contain, in addition to XML checks, tests of the functionality of the students' source code, so in the following testing we will use several tools. The question arises as to how the test structure should look. Typically, test cases are divided into several layers. For Android apps, the following test layers are common [27]:

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

As a strategy for automated testing we chose the concept of so-called *test pyramid* [30]. It says that any automated testing infrastructure should have more low-level unit tests than high-level UI tests. The concept includes the following 3 tiers (representing the pyramid):

- 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). 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 the listed test pyramid tiers we compared the various open-source test tools so that we could automate the testing of most student application functionality. We also consulted with *Wirecard* company. For unit testing, Google recommends *JUnit* with *Mockito* [31] tool to create mock objects (needed for tests where partial application context or an external service is needed). The purpose of unit test is to isolate the smallest source code units and check that, even in isolation, they work correctly. In Android applications, often even the smallest tasks are dependent on the app context, in which case *Mockito* can help us. In our case, we will most often test helper classes used to convert units, format text, generating alerts, and other similar functionality.

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. Using the aforementioned framework, one could also partially test user interface without a real device or emulator, with JVM only. The *Android SDK* is mocked by the framework.

For the last layer, UI testing, choosing a test tool was not as simple as the previous ones. Due to the presence of many competing UI testing tools with similar features, we had compared them in an earlier study [32] in 2017. We have evaluated these tools from a practical point of view, and in particular from the point of view of their testing capability. The most popular tools were *Appium*[4], *Robotium*[5] and *Espresso*[6]. However, 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 integration, support of OS vendor, emulator vs. real device testing, etc. The best tools from this point of view, as well as the most popular, 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 on the app UI).

4.2 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 executed very quickly.

4.2.1 Unit tests

Firstly, we started writing unit tests to get them as much as possible. In the sample app proposal, methods for cal-

3 http://robolectric.org/
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

culating calories, and changing the units of distance (km, mi), weight (kg, lb), etc., were precisely defined. Nevertheless, after a short while, we discovered that we had actually tested everything possible. Altogether, six different unit tests were made. This was problematic because the notion that the number of UI tests would be even smaller was unrealistic. Nevertheless, we maintained the chosen concept and tools.

Some tests need to be developed in conjunction with the *Android Software Development Kit* (SDK), but we did not wish to deploy the application to to an emulated 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 (where it was possible to to execute an 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 discovered that problems occur with every new version of the *Android SDK* when a new implementation of stub methods was 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.

4.2.2 Integration tests

A great feature of *Robotium* was an ability to automatically clear the testing 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, a question arose regarding how to test the intention of individual activities or starting/stopping app services. The desired action (new activity, system application request, etc.) is expected to be executed from the student application 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 of the *Robotium* tool were performed. All unit and integration tests are able to run using the *gradle*[9] tool.

We knew that, after receiving a specific action, a particular service should be started or stopped. However, we could not test the functionality of the service, which was a major part of the *Makacs* logic (calorie counting, distance, etc.). Despite the fact that *Robolectic* allows testing of service functionality, the solution was hard to implement. We found the *AndroidJUnit4*[10] tool which, while much easier to implement using a binder, required an Android device (virtual or real). Together we created 6 tests for the main service.

4.2.3 UI tests

At the highest layer of testing environment was UI testing. *Espresso* is a young test framework, which manifests to the developer because writing 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, therefore identifiers are required to be set manually. However, it's faster to modify recorded tests than writing all of them 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 a specific use case, e.g. data recovery from the 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 a particular test.

As part of the UI testing, we used the stress testing tool *Monkey*, which runs 10,000 random gestures for each student's solution to get the application into an unexpected error. We focus on 2 main issues: an unexpected app crash and unresponsive application.

8 http://ormlite.com/

9 https://gradle.org/

10 https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html

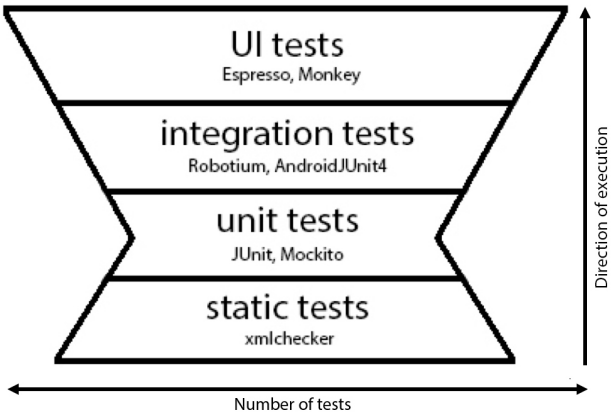

Figure 1: Tiers of proposed testing environment.

4.2.4 Final environment structure

Figure 1 shows the actual proposed structure of the testing environment. As can be seen, the testing environment in terms of the number of tests is the opposite of the test pyramid concept. As we did not wish to unnecessarily restrict the students' solutions in the assessment of their assignments, we needed to focus on testing that is independent of 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 we need a different amount of tests in individual tiers.

4.3 Course adaptation

All mentioned suggestions were prepared for the fall 2017 semester of the course (full results presented in Section 5). To validate the assumption that the testing environment is capable of evaluating assignments correctly, we performed a number of possible tests on a random sample from 2015 and 2016 runs of the course. Some results of manual testing have been already described in Section 3.1, based on which we focused on selecting test tools and implementing particular test cases.

4.3.1 Retrospective testing results

From the proposed test tools, student solutions from 2015 and 2016 could be tested only by stress testing using the *Monkey* tool, which is independent of the specific implementation of the solution. In order to use other tools or tiers of the environment, each student solution should have the required identifiers that the tests could refer to. Since the design of the testing environment was not known at the time of defining the assignment, it was not possible to concretize it. Due to the independence of the *Monkey* tool with respect to IDs we tested 10 randomly selected applications that were approved by manual teacher testing, where 50% of tested applications failed. In the test 10,000 random gestures were used for 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 are expected to test hundreds of the same applications several times a day, which could 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 2), 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), we expected our system could complete its checks on one real device in 6 hours 27 mins 30 sec, on average. As this is quite a long time, we planned to run tests every 12 hours.

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 until a few weeks later. However, this issue still persisted on some devices. We discovered that the problem occurred while animations on a device were enabled, so it is necessary to turn off all device animations before running tests.

4.3.2 Syllabus changes

From restrospective testing, we were once again more experienced in what direction we should take and how to avoid known issues. Because we expected that new issues would arise as soon as the testing environment was de-

Table 2: Execution speed of 71 tests on different devices in 10 measurements.

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

ployed, we decided to run tests only in the background, in essence without the impact of test results on a real student assessment.

From the viewpoint of implementation, the course changed in that the individual parts of the assignment were split into sprints, according to the *SCRUM* agile method. Based on this, the parts of the project were divided into 3 sprints:

- Sprint 1: GUI design.
- Sprint 2: base functionality (required by us).
- Sprint 3: extended (own) functionality.

In Sprint 3 (extended functionality), we gave our students the freedom of creativity and each student chose their own type of application extension. Therefore, this part could not be tested automatically. However, in this way, we wanted to see how the student code created in previous sprints would be affected. Thanks to the sprints, the tests could be divided into several runs. The requirement for each run was that the implementation of the new functionality should not affect the requirements of the previous iteration (except for the 3rd sprint).

As mentioned in Section 4.3.1 the plan was to run the tests every 12 hours. However, after deeper analysis we decided to choose an approach where the student would only have the assignment definition and the tests would be run at the end of the sprint/course. From this point of view, the approach is closer to manual testing when the teacher evaluates the assignment only at the end of the solution, thus the student would base his work only on the definition of the assignment and has no test results. This will allow us to better compare the results of the manual testing with the results from the testing environment. In automated testing, the student is accustomed to achieve test results on a continuous basis, which is very similar to the so-called test-driven development. Our experience is that if the student has test results available continuously, he or she is programming against the testing environment and stops thinking about the implementation and consequently thinks much less.

Assignment submission was implemented through the version control system (VCS) *GIT*[11] in cooperation with the *GitLab*[12] service. Based on the deadline of a particular sprint, we were able to use the *GitLab API* to fetch the last committed student code to the deadline and test the submitted version to meet the definition/requirements.

5 Results

In fall 2017 the course was attended by 132 students with an average final score of 70.66% (manual testing). Student source core was taken from the *GitLab* system on 1st February 2018, when 95 students submitted the assignment. Static testing included 36 tests, and average test failures in particular sprints are shown in Table 3. As we can see, between the 1st and 2nd sprints, the results improved slightly, because the 2nd sprint followed on the previous one and following tests required the presence of all mandatory UI elements. In the third sprint, students were made aware that automated testing would no longer be used in this iteration, which greatly reduced the static test success rate due to the implementation of their own extension. Because the extension should not interfere with existing functionality, and especially with design elements and their identifiers, students in the next course run will be warned to avoid it. The time for checking all students' projects was 63 seconds on average.

Table 3: Average results of static tests in individual sprints.

Sprint	Avg. failed tests	No. of tests	Avg. failures in %
1.	3.73	36	10.35%
2.	2.59	36	7.19%
3.	5.17	36	14.36%

11 https://git-scm.com/

12 https://about.gitlab.com/

Table 4: Static testing results.

Error rate		Number of students		
From	To	Sprint 1	Sprint 2	Sprint 3
0.00%		30	25	13
0.01%	10.00%	43	50	50
10.01%	20.00%	8	12	13
20.01%	30.00%	4	6	6
30.01%	40.00%	5	1	6
40.01%	50.00%	0	0	0
50.01%	60.00%	1	0	0
60.01%	70.00%	1	0	3
70.01%	80.00%	0	0	0
80.01%	90.00%	0	0	0
90.01%	100.00%	3	1	4

Table 5: Results of unit, UI and stress testing in 2nd sprint of 25 filtered solutions.

Testing type	Failed tests			Total tests no.
	Best	Worst	Average	
Unit	2	11	6.48	12
UI & integration	3	35	16.32	36
Stress	0	6	0.76	10000 gestures

Table 6: Execution times of tests.

Testing type	Time		
	Best	Worst	Average
Unit	11.98 s	42.89 s	26.07 s
UI & integration	24.88 s	71.04 s	44.95 s
Stress	33.94 s	86.30 s	53.64 s

Overall success in static tests during individual sprints and the distribution of students into intervals according to test failure is shown in Table 4. In the 2nd sprint, the testing environment did not only perform static tests, but also ran all the tests suggested in Section 4. Because we refer to specific UI element identifiers in the integration and UI tests„ we strictly insist on a zero error rate of static tests in the 2nd run to perform the other tests. Zero error rate means the student's source code includes all required elements and IDs for future testing. This requirement was only met by 25 students by the deadline of the 2nd sprint, so we continued to work on these projects only.

At the first stage of the testing process we have come across a problem with executing the student's code, which required a compilation.. Android uses build automation system *Gradle* to ensure build of application with different dependencies. For running tests we needed specific dependencies and students needed their own. The problem with the *Android SDK* and the *Gradle* version was that they often contradicted each other, leading to build dependency conflicts.. Conflicts between students' dependencies and those of ours have also often arisen. For *Gradle* files editing we used the UNIX *sed* tool to adjust the *Gradle* file to our requirements. However, most projects still needed our manual intervention. Therefore, in the future, it will be better to provide a default *Gradle* file to students with our dependencies and the student must ensure that the application is compilable with the required settings.

The results of the unit, UI, integration, and stress tests can be seen in Table 5 and their execution times are in Table 6. The total duration of the tests was 52 min 57 sec. Execution times do not include compilation and *Gradle* build time. While performing unit tests, 2 projects could not be tested due to incorrect names of packages, so it was necessary to manually modify the project. In a normal automated process, we do not edit students' projects, but now we tried to get the most results and discover the most unidentified errors in the testing environment. Testing failed even though the student code contained logging because `Log` java class was not mocked. That's why we've created our own mock implementation because logging is not a reason for testing failure. The last issue was the missing methods referred to by the testing environment. But this is a common matter of testing of students' solutions. Table 7 details test results for each student separately.

Performing UI and integration tests was slightly more challenging. As suggested in Section 4.3.1, we used the Prestigio 5453 DUO, API 19, Android 4.4 real testing device. Since the number of tests in the 2nd sprint was about half of tests created for sample application this also reflected the length of tests execution (compare with Table 2). In terms of performance, average speed did not change, so running outside of an emulator was many times faster and more reliable. During the execution of the GUI tests `NullPointerException` occurred most often (14 times). The exception occurred repeatedly during `Timer` management, work with an intent's extra parameters and list of GPS coordinates.

We did not place obstacles to students when choosing a programming language, so we met with testing app developed in *Kotlin*[13]. We were unable to execute one of 3 test cases of UI tests because of `NullPointerException`. However, this behavior was not caused due to the choice of language, but rather due the wrong implementation. During

13 https://kotlinlang.org/

Table 7: Failed tests and assessment differences between manual and automated testing.

Student	Failed tests			Assessment		Difference
	Unit	UI & integration	Stress	Automated	Manual	
S1	8	35	1	9.42%	100.00%	90.58%
S2	2	24	6	40.83%	100.00%	59.17%
S3	9	28	0	22.92%	81.00%	58.08%
S4	8	13	3	53.25%	100.00%	46.75%
S5	7	15	0	54.17%	98.00%	43.83%
S6*	3	26	0	39.58%	83.00%	43.42%
S7	5	14	0	60.42%	100.00%	39.58%
S8	10	13	0	52.08%	91.00%	38.92%
S9	11	18	0	39.58%	75.00%	35.42%
S10	4	11	0	68.75%	100.00%	31.25%
S11	6	17	0	52.08%	81.00%	28.92%
S12	9	16	0	47.92%	75.00%	27.08%
S13	9	23	3	30.33%	52.00%	21.67%
S14	7	12	1	59.42%	81.00%	21.58%
S15	6	16	0	54.17%	75.00%	20.83%
S16	7	18	0	47.92%	68.00%	20.08%
S17	6	14	0	58.33%	78.00%	19.67%
S18	5	15	1	57.33%	75.00%	17.67%
S19	3	12	0	68.75%	52.00%	16.75%
S20	8	16	1	49.00%	58.00%	9.00%
S21	6	10	0	66.67%	60.00%	6.67%
S22	10	11	0	56.25%	52.00%	4.25%
S23	4	12	0	66.67%	63.00%	3.67%
S24	4	16	0	58.33%	60.00%	1.67%
S25	5	3	3	80.33%	81.00%	0.67%

Assessment: 100% = best; * - *Kotlin* as programming language

testing all applications, overall 5 test cases of UI and integration tests failed. We were not able to influence this fact because a unimplemented class or method that was necessary for testing was the most common error.

As part of the integration tests, we tested the application service as well. For us the easiest way to test the service was using a `Binder` class. The students had to implement a method with return of a `Binder` object; unfortunately, only 5 projects had the required method correctly implemented and usable. We had to manually edit the other projects to get more results. In the future, it will be better to provide a prepared service class with a binder to avoid uncovered problems.

Using the *Monkey* tool in real testing 8 applications failed, which is only 32% of tested projects. Compared with 2016 (Section 4.3.1), this result improved by 18%, probably indicating good course direction and student guidance when programming stable applications. We used a real device for stress testing for the first time and the execution was also 2-3 times faster than on the emulator (like with the UI tests, see Section 4.3.1).

As a final part of the evaluation of students' assignments we tried to identify the differences in the assessment between the automated environment and the manual assessment by teachers. These results are shown in Table 7 and are listed in descending order of difference between manual and automated evaluations. The total number of tests is mentioned in Table 5. The rating scale in the *Application Development for Smart Devices* is as follows:

- 91% - 100% = A
- 81% - 90% = B
- 71% - 80% = C
- 61% - 70% = D
- 51% - 60% = E
- <51% = FX

The rating scale shows that 10% of the assessment difference is significant for students, as this difference affects their final grade. Table 8 therefore shows differences after 10% intervals between manual and automated testing resulting from Table 7. As can be seen, the difference was less than 10% in only 6 cases, indicating a significant difference in automated and manual evaluation.

In our opinion, the difference was due to the way we examined the tests. In manual testing, the teacher examines the application from the perspective of the user (mostly in the way of black-box testing), and the implementation is rarely checked in detail. In automated testing, both viewpoints are covered: 1) UI tests verify the functionality of the user (black-box); 2) the remaining tests verify the correctness of the implementation (white-box). Automated tests are also able to test all the required borderline cases within a short time period, with the teacher taking a lot longer and being less reliable. Finally, in manual testing, we can not implement stress testing because no one is fast enough to perform it.

Differences in the assessment could also be caused by the fact that manual evaluation was performed by multiple teachers. Students have the same application requirements, but teacher's assessment requirements did not have to be the same, eg. for the first teacher general functionality could be more important; for the second, borderline cases; for another, implementation correctness, etc. The proposed test environment reduces this fact and ensures a fairer and more equitable assessment for all students.

As the results point out, it is not appropriate to use an approach without continuous feedback of the testing process, as the student can not sufficiently improve his or her solution. When a student does not see results of tests, he or she often cannot identify the mistake/issue in his/her code, his/her final grade will decrease, and he or she will not improve his/her programming skills. On the other hand, to avoid trial-and-error methods often observed while watching students, it will be necessary to find the right level of both approaches – sufficient but not excessive testing attempts.

Table 8: Quantification of assessment differences in 10% distribution.

Assessment difference	No. of students
0 - 10%	6
10 - 20%	3
20 - 30%	6
30 - 40%	4
40 - 50%	3
50 - 60%	2
90 - 100%	1

6 Conclusion and future work

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 testing environment and used the appropriate test tools for the proposed tiers.

We have succeeded in designing a new testing environment that is unique to its coverage. An *xmlchecker* tool has been created that can be used to statically test UI elements or XML files in general. The *test pyramid* concept has been customized for mobile devices testing which is the unique testing environment where it is necessary to test different implementations of the same requirements. The solution includes dynamic tests and proposes the use of multiple tools and their collaboration in the assignment assessment. At the same time, we evaluate the work with these tools to achieve these goals.

In order to get partial results and experience with testing and to design as good environment as possible, we continuously performed a retrospective stress tests of students' assignments from the last run of the course of using the *Monkey* tool. 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. Because it was not possible to use all the tests that we created for previous years' solutions, we tried to run them on the sample application. Based on experiences from the testing process, we suggested changes to the course and similar courses to use the testing environment in it.

Finally, we compared assessment differences in manual and automated testing. The main research sample was the 2nd sprint assignment submission, where all kinds of tests were deployed. The student could submit the assignment only once. Static tests were passed by only 25 projects out of the total number of 95, which we examined in more detail. Based on the results, we are looking for minor deficiencies in our massive testing environment of assignments, and simultaneously specifying the most common failures of the applications during testing in order to avoid them in the future. We also compare the performance of the tests to estimate their execution time in full deploy-

ment. The authors compare, among other things, the results of automated testing with manual evaluation, and confirm that manual testing is needed as well but it is highly subjective and does not create the same assessment conditions for students. Therefore, the authors consider automated testing to be more effective, fairer and more reliable.

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. Due to the identified deficiencies in the definition of the assignment and detected issues while testing, we plan to fix all issues and use the testing environment as a major evaluation tool in the fall 2018. We also would like to extend the tests for checking translations and different fixed values in the source code (such as colors, dimensions, etc.) in order to use them for checking runtime design changes or reference them for preforming other tests.

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, then 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. The evaluation of the tests made by the students is addressed by Smith et al. in the paper [33] from 2017.

Acknowledgement: 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] Meeker M., INTERNET TRENDS 2017, Technical report, CODE CONFERENCE, 2017

[2] StatCounter, Operating System Market Share Worldwide, 2017, http://gs.statcounter.com/os-market-share#monthly-201101-201708

[3] Esposito J., et al., Mobile Application Development, volume 3, DZone, 2016

[4] Cook M., The 50 Most Popular MOOCs of All Time, 2015, http://www.onlinecoursereport.com/the-50-most-popular-moocs-of-all-time/

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

[6] edX Inc., Android Courses Search, 2017, https://www.edx.org/course?search_query=android

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

[8] Domin C., Pohl H., Krause M., 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 (CHI EA '16), ACM, New York, NY, USA, 2016, 1173–1179, DOI 10.1145/2851581.2892512

[9] Mann S., Frew Z., Similarity and Originality in code: plagiarism and normal variation in student assignments, In: Proceedings of the 8th Australasian Conference on Computing Education - Volume 52, ACE '06, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2006, 143–150

[10] Munir H., Wnuk K., Petersen K., Moayyed M., 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 (EASE '14), ACM, New York, NY, USA, 2014, 50, 1–50:10, DOI 10.1145/2601248.2601267

[11] Madeja M., Porubän P., Automatic assessment of assignments for Android application programming courses, In: Novitzká V., Korečko Š., Szakál A. (Eds.), 2017 IEEE 14th International Scientific Conference on Informatics, IEEE, 2017, 232–237

[12] Mondal S., Vrushabendrappa L., Android Application Development, 2017, https://www.springboard.com/learning-paths/android/

[13] Schmidt D. C., et al., Launch your Android app development career, 2017, https://www.coursera.org/specializations/android-app-development

[14] Catalán A., Branagan N., Anzueto F., Professional Android app development, 2017, https://www.edx.org/course/professional-android-app-development-galileox-caad003x

[15] Pecinovský R., Zadávání a vyhodnocování úkolů při výuce OOP, in Počítač ve škole 2007, Amaio Technologies, Inc, Počítač ve škole, Nové Město na Moravě, Czech Rep., 2007, 1–4

[16] Sugrue J., et al., Automated testing, volume 1, DZone, 2017

[17] Souce Labs, Mobile app testing: Main challenges, different approaches, one solution, DZone, 2017

[18] Adamsen C. Q., Mezzetti G., Møller A., Systematic execution of Android test suites in adverse conditions, In: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015), ACM, New York, NY, USA, 2015, 83–93, DOI 10.1145/2771783.2771786

[19] Amen B., Mahmood S., Lu J., Mobile application testing matrix and challenges, Computer Science & Information Technology, 2015, 5

[20] Akourm M., Falah B., Al-Zyoud A. A., Bouriat S., Alemerien K., Mobile software testing: thoughts, strategies, challenges, and experimental study, International Journal of Advanced Computer Science and Applications, 2016, 7(6), DOI 10.14569/IJACSA.2016.070602

[21] Wang Y., Alshboul Y., Mobile security testing approaches and challenges, 2015 First Conference on Mobile and Secure Services (MOBISECSERV), IEEE, 2015, DOI 10.1109/MOBISECSERV.2015.7072880

[22] Muccini H., Francesco A. D., Esposito P., Software testing of mobile applications: Challenges and future research directions, In: 2012 7th International Workshop on Automation of Software Test (AST), 2012, 29–35, DOI 10.1109/IWAST.2012.6228987

[23] Gupta S., Dubey S. K., Automatic assessment of programming assignment, In: Meghanathan N., et al (Eds.), ITCS, SIP, JSE-2012, CS & IT 04, 2012, 315–323, DOI 10.5121/csit.2012.2129

[24] Wilcox C., Testing strategies for the automated grading of student programs, In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16), ACM, New York, NY, USA, 2016, 437–442, DOI 10.1145/2839509.2844616

[25] Rajala T., Kaila E., Lindén R., Kurvinen E., Lokkila E., Laakso M. J., Salakoski T., Automatically assessed electronic exams in programming courses, In: Proceedings of the Australasian Computer Science Week Multiconference (ACSW '16), ACM, New York, NY, USA, 2016, 11, 1–11:8, DOI 10.1145/2843043.2843062

[26] Akahane Y., Kitaya H., Inoue U., Design and evaluation of automated scoring: Java programming assignments, International Journal of Software Innovation, 2015, 3(4), 18–32, DOI 10.4018/IJSI.2015100102

[27] guru99.com, Complete Guide to Android Testing & Automation, 2017, https://www.guru99.com/why-android-testing.html

[28] Wasserman A. I., Software engineering issues for mobile application development, In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10), ACM, New York, NY, USA, 2010, 397–400, DOI 10.1145/1882362.1882443

[29] Lewis W. E., Software Testing and Continuous Quality Improvement, Third Edition, Auerbach Publications, Boston, 3rd edition, 2008

[30] Greb C., The 3 tiers of the Android test pyramid, 2016, https://medium.com/android-testing-daily/the-3-tiers-of-the-android-test-pyramid-c1211b359acd

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

[32] Madeja M., Testing of applications for OS Android, Master's thesis, Technical University of Košice, 2017

[33] Smith R., Tang T., Warren J., Rixner S., An Automated system for interactively learning software testing, In: Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17), ACM, New York, NY, USA, 2017, 98–103, DOI 10.1145/3059009.3059022