Evaluation and Qualification of Mobile Application Quality

Rudy Bisiaux, Mikael Desertot, Sylvain Lecomte and Dorian Petit
Univ. Polytechnique Hauts-de-France CNRS, UMR 8201 - LAMIH, F-59313 Valenciennes, France
Keyneosoft ,31 Rue de la Fonderie, 59200 Tourcoing, France

Development of mobile applications has previously encountered, and still encounters, challenges related to the specificity of the mobile world. The usual rules of software engineering tend not to respond to certain mobile development issues. In this paper, we identify and address the challenges of mobile development. We propose a quality model and analysis adapted from ISO9126. Based on this model, we have developed a tool to analyze an applications source code and measure the achievement of quality criteria. We describe all the notions of quality and the check points in the mobile applications source code. This work was carried out in partnership with a mobile application development company, Keyneosoft, to test our tool on real-world applications. We also discuss the results obtained and feedback from Keyneosoft developers on our tool

Keywords: Mobile applications, Software quality, Development process, Software Components

1. INTRODUCTION

The field of mobile development is a recent sector of computing and has a number of functionalities that can affect the quality of applications offered to users. The concept of mobile development affects different media: hardware, such as smartphones, tablets or watches, and software, such as cloud computing. Development of mobile applications is not subject to the same constraints as the development of "conventional" applications given the context of development, the proliferation of applications on the market, and especially the challenges to be met.

The deadlines for developing an application are short compared to a conventional application: they are counted in weeks for the production of an application. In addition, once the initial development has been completed, the application continues to evolve regularly to add functionalities. The frequency of updating an application contributes, more than in conventional development, to preserve a feeling of stability for the end user. These functionalities are added as modules, which must maintain a level of quality at least equal to that of the previous version. Short delays and modular development are not the only constraints to develop a mobile application. One of the additional problem factors when developing an application is the number of people who can use it. If the goal is to reach a maximum number of users, the systems and devices on which the application can be deployed must be multiplied.

Historically, developers have had to adapt to up to four different systems, but currently, two major companies share the monopoly of mobile systems: Apple with iOS and Google with Android. Microsoft's Windows Phone and BlackBerryOS have recently been dropped for the general public. These two operating systems (iOS and Android) are based on a different kernel, language, and development environment. Duplication of systems requires doubling the teams or the application development time, which represents a significant cost. The versions of these systems also evolve each year, which requires developers to constantly evolve the application code so that it remains compatible with the operating system and the different APIs proposed. Responses are currently under development, the main one being the use of cross-platform frameworks. Each

This research was carried out as part of a CIFRE contract managed by the ANRT (the french National Association of Technical Research) between LAMIH (https://www.uphf.fr/LAMIH) and KEYNEOSOFT (https://www.keyneosoft.com/), a company whose core business is to digitize a sales point and help salespersons streamline the customer's experience.

platform has different types of devices: from TVs to watches, to tablets and especially smartphones. This fragmentation must consider all the screen formats on which the application will be available. The large number of different devices forces developers to consider high usability, image size, and even a different functionality for mobile applications.

Added to this is fragmentation of the manufacturers. This phenomenon, only on Android, is caused by the willingness of manufacturers to modify and adapt the device environment, which requires once again the developers to take this into account during development.

Development time, the need for constant evolution, and the heterogeneity of operating systems and hardware are all important constraints on application development, which are rare in the context of so-called conventional computing.

We have highlighted the different challenges for the development of a mobile application and emphasized the costs of these challenges, but one of the biggest sources of cost is quality. From the start of development, cost must be considered. The various constraints mentioned above are factors that strongly influence software quality. Notions of quality and how to measure it in conventional computing are not necessarily adapted to the mobile application context. As context is an important part of mobile environments, we want to find a way to guarantee that our applications are robust and perform according to context variations.

When, for example, a smartphone loses its connectivity, the application must not be stopped accidentally, block the user, or lose the data it was processing.

These conditions of resistance to context variation are still poorly defined in the current design of a mobile application.

In this paper, we describe the framework of experimentation of these works and the state of the art on the challenges of mobile application development. We present our proposals to improve the quality of software development for mobile applications and the results of the evaluations conducted jointly with KeyneoSoft.

2. WORKSPACE

This research was carried out as part of a CIFRE contract between LAMIH and KEYNEOSOFT. The company, created in 2007, specializes in the development of mobile applications. Working closely together allowed us to access multiple mobile applications with a core of common source code. These applications are cross-platform applications developed in Xamarin (Dickson, 2013). Xamarin 2.0 is a Mobile Application Development Framework, originally created in 2013. It aims to share the code of a product application in Csharp and then render it compatible with the target platform, Android or iOS, using monoTouch.

The applications were designed based on the component model [Cai et al., 2000] to maximize the reuse of the core components [Perchat et al., 2014] and follow the MVVM (Model-View-ViewModel) design pattern [Syromiatnikov and Weyns, 2014]. The design pattern allows separation of the view and the business code, which gives the application more flexibility with respect to the type of platform, its version, or its support. We defined our interface using views (VIEW). Data were stored and manipulated in the models. Data transmission and navigation were provided by the viewmodel. The applications were built and ran using a continuous integration platform. The logs and source code were analyzed to assess the quality.

3. STATE OF THE ART

3.1 Mobile code production

Firstly, we discuss the so-called conventional industrialization points of software, point out their weaknesses, and describe the issues we wish to address. The first point to address is the software production line (SPL) [Hallsteinsen et al., 2008], defined by the Software Engineering Institute (SEI), to manage and organize software product processing. It is a set of software engineering methods for creating an application from a set of software resources to design and build easily maintainable and scalable software while minimizing errors that generate significant costs.

A software production line is divided into four main stages of production.

—The functional design defines the functional scope of the software. The functionalities, the behavior when used, and the design of the software are defined before development begins.

—The technical design defines the software architecture by enumerating the dependencies between the elements and by aiming the best technical solutions to respond to the software needs. During this phase, the libraries that will be added to the application are chosen. Libraries are often produced by the developer community or a hardware publisher, which sometimes makes it difficult to access the library's source code. Similarly, mobile applications are broken down into modules by following the component model. As a result, business functionalities are regularly isolated in components to enable their reuse across multiple applications. Unlike libraries, components are often produced by the application's publisher, which reuses and improves them. The publisher also produces unit tests to ensure the proper functioning of the components. These technical and functional designs are made by experts. The components designed during this phase are then produced during the development phase.

—Realization is the stage where the software development is achieved. The software is developed following certain rules specific to the type of software desired.

—The last step is the test and validation phase to ensure the stability and reliability of the software.

As described above, the running environment of a mobile application is particular. It is highly dependent on context, connectivity, the GPS signal, and other hardware components of which the status changes regularly. If an application depends on these components, it must always be able to work at its best, regardless of the situation [Popovici et al., 2012, 2011].

More and more mobile applications also require external data, either via web services or databases. Depending on the data volume, the application can retrieve them if necessary or store them locally so as not to depend on the Internet connection. The mobile application must store data without losing or altering them.

The test phase aims at validating the different components and functionalities created during the implementation phase. There are different types of tests: unit tests, integration tests, user interface tests, and functional tests. At the end, the delivery phase distributes the final product.

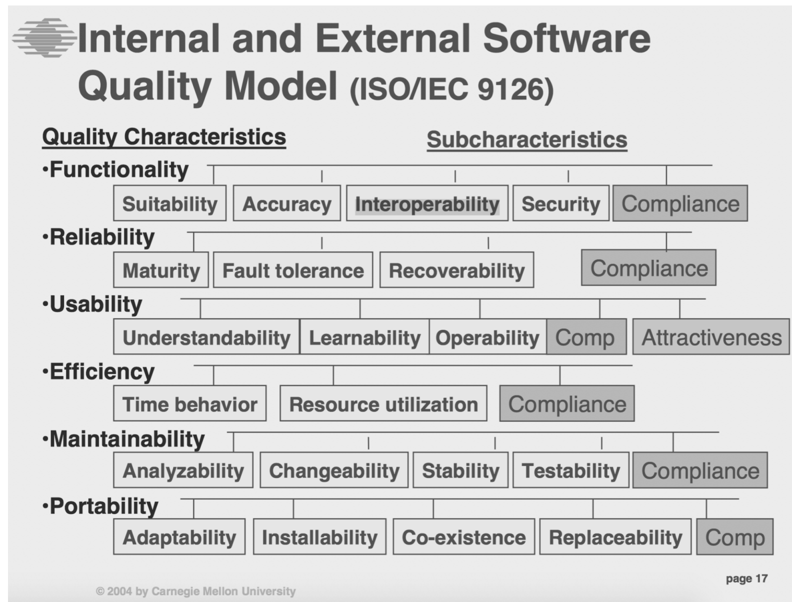
The Continuous integration is the usual way to automate these phases [Duvall, 2007]. With a few tools, we can automate some parts of the software process. An orchestrator reads defined tasks to control source repositories or source code with different versions, compile code, run tests, and deliver. These processes are associated with team management methods. Team management can impact the process quality. Team size and development time led us to use the Scrum management method [Schwaber and Sutherland, 2017]. This method is particularly well adapted to the characteristics described in extreme programming. In addition to team management, numerous standards must be followed to ensure quality in software development, including observable and adaptable design patterns, such as MVVM, MVC, Singleton, as explained in [Franke and Weise, 2011].

3.2 Concept of Quality for mobile applications

Our research focuses primarily on the quality of mobile applications. Quality is an important topic for software. Although many solutions exist for conventional applications, mobile constraints are not integrated into these solutions. The need for a specific mobile quality model is evident, as described in [Dehlinger and Dixon, 2011]. Software quality is a vast subject already validated by different types of certifications.

Certifications exist based on the software company's structure, such as CMMI [Team, 2006]. This approach assesses the maturity of the company to determine its ability to produce quality software. This approach was not initially adopted because it is organizational, whereas our approach focused on the quality of the developed product. Moreover, this approach is not suitable

for mobile applications [Joorabchi et al., 2013] (due to the fastest way to deploy this kind of applications). Production certifications exist, such as the most popular SQuaRE, ISO 9126 certification [Zubrow, 2004]. Application quality is defined by evaluating different criteria built into the meta-criteria, as described in Figure 1. We distinguish six meta-criteria that evaluate software quality. These meta-criteria are divided into sub-criteria to better describe the expectations of each one. For all the criteria defined here, we search for their representation in the software's source code, i.e. a notion is interpreted to find one or more representations in the source code. For example, the notion of robustness can be interpreted by a strong resistance to the sudden shutdown of the application that can result in a try / catch regular implementation in the code or a nullity test on the objects. In our approach, we aim to validate these criteria by using the



1 - Description of the square standard

mobile context control points. When Square certification renders a criterion, we need to find a way to verify it during mobile development. Some approaches define a mobile quality model. For example, [Zahra et al., 2013] proposes to add data integrity notions to validate their consistency when the application is paused or stopped, but this is insufficient. Also, [D. Garofalakis et al., 2007] propose a new definition of quality criteria for a specific branch of mobile applications: The M-commerce. Because business logic distribution is heavily used in mobile software, the verification of data integrity must be constant when the application interacts with other systems, such as servers. We need to generate control points that match the Square certification quality criteria for each component of a mobile application.

4. MARKERS TO ENSURE QUALITY

We aim to ensure the quality of an application throughout its development cycle. To do so, we first ensure that the application respects previously produced designs. We then ensure the quality of the code, libraries, their integration, and consider the specificity of the mobile world. The test coverage must be sufficient, the tests implemented, and the results valid.

All these points come at a cost in terms of design, implementation, and application maintenance. The quality of a mobile application must be sustainable throughout its development cycle.

At first, it is interesting to compare the development of a mobile application and that of a so-called conventional application (server, desktop, or web).

In general, industrialization and quality assurance are intimately linked. The latter is particularly time-consuming because tests are run, code is reread, and the application must works properly every time a change is made. In the production of conventional applications, some of these steps are executed automatically. Generic validation criteria inherent to the language or algorithmic are used, for example, running tests or compiling. This is a little different in the mobile world because of the great heterogeneity of hardware and software. We have different platforms, different libraries, different media, and a context that can change at any time. A mobile application must adapt to all these types of changes.

As mentioned above, functionalities, such as adapting to changing context or persisting and / or recovering data in real time, are challenges we must address and for which we must ensure the quality of execution.

The first step in our research was to evaluate these needs to define the quality criteria (or markers) that validate our mobile application and then make a prototype to search and validate these criteria in the source code of a mobile application [Bisiaux et al., 2017]. We then evaluated the relevance of the feedback from developers on our prototype.

What we call mobile application-specific criteria is a list of markers used to determine whether a mobile application is of high quality or not. These markers, as mentioned earlier, are verified by static analysis of the code. Throughout the development cycle, we validate these markers one by one to ensure the quality of a mobile application. Markers, such as the SPL development cycle, are divided into four groups.

—For the design, we aimed to ensure that the product code conformed to the technical design realized by the architect.

—For the realization, we aimed to ensure that the code was both of high quality and took into account the specificity of the mobile world. Depending on the target platform, the SDK, and the support, we used markers able to ensure that certain obligations are implemented.

—For the tests, we validated the test coverage of the application and ensured that all tests are passed.

—For the delivery, we aimed to validate the applications quality after archiving it. Subsequently, the signature and obfuscation were ensured.

4.1 Description of markers

We relied on the development of a mobile application to detect compilation, test, or maintainability issues. To do so, we defined markers, which represented a formalization of notions associated with quality in the mobile world. These concepts are the ones we have mentioned in the software quality standards section. We aimed to ensure the functionality required when defining the application, code robustness, maintainability, usability, and scalability of our mobile application. Markers associate a notion of quality with different evaluation methods in the context of mobile development. We classed our markers into different categories that bring together the different assessment methods of each requirement to allow our tool to trace errors at different points in the software production cycle.

Category A markers intend to allow analysis of the various tests, and build reports of the mobile application and the dependencies it requires to function. A mobile application regularly links to libraries to use code already produced. These libraries have interfaces to facilitate the use of various materials, such as EPT (Electronic Payment Terminals), printers, Bluetooth devices, or API development software. This avoids errors by recreating code already produced and achieving higher level functionalities through the expertise of the developers producing the library.

Category B markers allow static analysis of the application's source code. This analysis focused primarily on good code development practices. These good practices are general, but we adapted some of them to meet the needs of mobile development, and sometimes even for the specificity

of Android development. We published markers that aim to limit issues related to the maintainability, robustness, and efficiency of the code by searching for problematic code occurrences during the production of the mobile application code.

Category C markers, like category B markers, allow the static analysis of the application's source code. They focus mainly on context adaptation. We aimed with the markers of this category to ensure the presence of code to certify that the use of these elements is verified and that the context is favorable to it. Take the example of connectivity: what happens if an application that needs to interrogate a remote server is not connected to the Internet? Should it wait, or warn the user and shut down? We defined markers that make sure connectivity is tested before making a call to a remote server. This limits the errors during the application execution and improves the applications robustness against context changes.

Grade D quality markers aim to ensure what we call continuous validation. The previous categories of markers ensure the application quality at the time of production of the mobile application source code; nevertheless, some errors cannot be anticipated at the time of development. These errors are related to either performance issues or simply malfunction of the application. Malfunctions can be varied, such as sudden shutdown, overruns, or memory manipulation errors. The markers that we defined are a non-exhaustive list of potential performance issues encountered when using the application in production.

4.2 Test markers

As defined above, category A markers are primarily related to tests execution. Different types of tests exist and we distinguish three of them. The first type corresponds to unit tests that focus exclusively on the validation of small pieces of code often limited to one method. A unit test validates the operation of this method. To do so, it needs input parameters and expected results. Once these elements are defined for each unit test, they are executed by a developer or automatically in the context of continuous integration.

The second type of tests corresponds to functional tests that focus on the validation of the functionalities defined in the analysis phases. These tests therefore contribute to the validation of the methods operation and ensure the proper use of them.

Recently, we obtained integration tests. They test the use of the libraries or components that we presented previously. If we take the example of a library, an integration test ensures that the library is used correctly to avoid errors when using it. This test is the most complicated to set up. It requires a recording platform to be able to realize application use scenarios. The platforms are sometimes not available for all environments targeted by cross-platform development. The tests validate the operation of the various functionalities of a mobile application. Scenarios are played out on the mobile application, for example, creation of a customer account. Clicks, navigation, and information are recorded during the first use by a human operator. Once the scenario is saved, it can be replayed automatically when running the tests. This also makes it easy to test on different devices. Use of these tests ensures that dimensional constraints, for example, always allow the use of the application regardless of the capabilities of the terminal used. In our work, we do not focus on this type of test for the realization of our markers because their use is different from the two previous ones. The tests will be part of future perspectives of our tool's evolution.

The list of all possible unit and functional tests in the mobile application's code remains to be established. This list should be as exhaustive as possible in order to achieve maximum coverage. Execution of the tests highlights the errors produced during modification of the source code. These errors can be of multiple sources, i.e. the modified code is in question or the test needs to be updated. In our quality approach, we aimed to identify errors to highlight them and facilitate their resolution. The different markers associated with the validation of the various tests are described below.

—A0: Libraries unit test: If we have the source code of the libraries used by the application, we collect data on the execution of these unit tests.

—A1: Unit testing of components: Like libraries, the different components external to the application can be tested and evaluated.

—A2: Functional tests: These tests evaluate a complete functionality, such as field capture or navigation.

—A3: Library integration: Once a library is tested, we need to make sure that it is correctly implemented.

—A4: Integration of components: In the same spirit, when using external components, calls such as returns can be tested upstream to anticipate potential problems and especially locate them more easily.

—A5: Application: The application must also be validated manually. A file can be edited by following an acceptance plan to target errors related to use.

4.3 Markers related to code quality

This category focuses on markers related to code production. We aimed to ensure code quality by these different markers via an analysis of the source code product. The definition of these markers is based on the expertise of Keyneosoft developers. We also have markers specialized in the detection of adaptation code specific to the heterogeneity of mobile applications (system versions, etc.). We have a list of codes that regularly pose a problem when rendering a mobile application compatible with several versions of the operating system. This list is also based on the expertise of Keyneosoft developers, but also on the study of operating system release notes provided by Google. The release notes list the modifications necessary to make the applications compatible with the new versions of the operating system. Once a problematic code is detected, we search for compatibility provisions that have been considered elsewhere in the application's source code. If these are present, then the marker is validated; otherwise the marker is invalid. Invalid markers appear in the final report of the execution of our tool. We describe below a list of markers that impact the different notions of our quality model.

—B0: Design / Retro-Design: Aims to be compliant with the design. We can perform a retro-design on the existing code to ensure that it was consistent with what was produced in the documentation during the design stage.

—B1: Design pattern: Aims to detect the design patterns used in the application code to make sure they meet their standards [Pree and Gamma, 1995].

—B2: Code duplication: Aims to limit this because it is source of error, particularly when changes are made to one of the duplicates. By limiting it, we can minimize possible errors.

—B3: Code interdependence: Searches for redundancy of call to components / libraries. Sometimes the application calls a component that calls the application itself to recompose the component recursively. This happens regularly because of the event aspect of a mobile application. We want to limit it to the maximum.

—B4: Software resource management: Aim to ensure the management of software resources, such as files or certificates. If a file is used, we want to make sure that the code ensuring the presence of the file exists in order to limit errors during execution.

—B5: Hardware resource management: As for software resources, the use of hardware must be tested to ensure that it is available. If the code allowing this test is not present, we consider that the potential errors are critical because they regularly cause the sudden shutdown of the application.

—B6: Permissions management: One of the critical points in Android is the use of permissions. These are mandatory in the release, but not in the debug, which is the source of regular oversight during development. The marshmallow version of Android forces the request for permission while running the application. For each use requiring permission, it must be requested.

—B7: Notifications management: Push notifications evolve regularly on each platform and their specific implementation requires a special evaluation.

—B8: Deep link management: Aim to organize a diagram related to deep navigation in a mobile application. Currently, it is done manually. Applications can have multiple entry points, accessible via pre-recorded links in the application to allow navigation to a specific functionality of the application.

—B9: WebViews management: This requires specific implementation and navigation for which we must ensure that they are correctly implemented.

4.4 Markers related to contextual adaptation

We aimed to adapt our mobile application to context changes. We described above that the mobile world is highly context-dependent. What we call context is the hardware and software environment of a mobile application. Developers know that they do not fully control the behavior of their application. If the user decides to pause, stop, or restart the application, this type of event must be considered. In addition, applications today also depend on other fluctuating software and hardware resources, such as the mobile network, the WIFI connection, and GPS data. An application requiring this type of resource must ensure its robustness when faced with changes in availability. Context adaptation [Coutaz et al., 2005] is related to the notion of robustness in our mobile quality model. We have therefore published markers that our tool uses to ensure the presence of the necessary code when it detects the use of a resource that may be sensitive to context changes. For example, our tool detects the presence of code using Internet access by the code pattern in the markers. We search for the source code to test the device connectivity. If this code is in the application, the marker is valid; if not, the marker is invalid and is centralized in the list of invalid markers. The list of markers in this category is described below.

—C0: Connectivity test: Ensures the presence of all types of connectivity, WIFI, cellular networks, Bluetooth, RFID, etc.

—C1: Memory leak test: Ensures that the application does not cause memory leakage by a mishandled allocation.

—C2: Background tasks test: Browses the services to make sure that they are stopped at the right time.

—C3: User interruption test: Ensures that the life cycle of an Android mobile application is respected.

4.5 Markers related to continuous validation

We wanted to ensure the quality of the application in the continuity, even after the start of production. The idea was to apply trackers on some markers to ensure the quality of the application, which required adding additional code to the application's source code to measure the different points that interest us. This measurement can be carried out on the entire park used and makes it possible to realize averages of execution time, for example. The markers ensure quality traceability even when the latter is no longer in development, and also when it is used in production. Here is the list of markers that interest us.

—D0, Crash Analytics: Analysis of sudden application shutdown feedback. This helps to determine the cause of a shutdown.

—D1, Web performance: Analysis of the performance of calls to the web service.

—D2, Database Access Performance: Database access performance analysis.

—D3, Hardware Performance: Performance analysis of the processor, memory, or sensors (GPS, Gyroscope, etc.).

—D4, WebView Performance: Analysis of the WebView loading performance.

5. MARKER EVALUATION METHOD

To validate the quality using criteria specific to the mobile world, which we defined, we used an application from Keyneosoft. The main application on which we evaluated our work was

Keysales; this mobile application, considered a product: a product as defined by Keyneosoft is a customizable application adapted to the customer's needs. Each customer uses the same application, but it is customized through by parameters to make it unique. Inherently, Keysales is a single application. If an additional functionality is added to the application, all other customers can use it. This system imposes particularly drastic monitoring of the application quality. Some functionalities may compete and the addition of these can cause regressions. If two customers need a similar functionality, but their operation is not shared, they must not appear at the same time in the application.

In addition, as the application is compiled for different customers, tests must be performed on the same application, but under different configurations, to ensure that an application delivered to a customer is not affected by its specific changes.

5.1 Prototype and integration

We implemented a code analysis tool written in Java. Java was chosen by convenience of the developer because there are no performance concerns to impute the comparison of strings in our solution. This tool is based on a pattern search (the markers defined above) in the application's source code. A unique structure allows us to represent all of our markers, which defines the list of patterns (pieces of code) linked to the markers. These pieces of code are termed "value". The associated required code that is specific to the target platform is also found in this structure. These pieces of code are the "should" and "must" codes. A category is also associated with each of the markers.

At first, the tool browses the entire applications source code and then compares it with the list of markers. The tool considers a marker active if it finds at least one of the marker's patterns in the applications source code when comparing it to the list of markers. Once a marker is activated, we try to solve it. To do this, a search for "should" or "must" codes is implemented. Any difference found assigns a level of criticality to the absence of the pattern. A resolved marker with only the source code in the "should" code is a marker of lower quality. Once a marker is detected and the quality elements are checked, we can move on to the next one. We described above that quality markers are divided into different categories. Each marker category is divided into different files to categorize resolved markers from unresolved markers and to add and modify the markers more easily. This approach guarantees the code's quality and maintenance by ensuring that there is no missing code and that the code is in the correct place.

As we described in the definition of the different markers, this method is used for category B and C markers. Category A markers are more related to the analysis of the application's source code data and the results of the execution of the various tests that we describe below.

Once all the markers are evaluated on the entire applications source code using the analysis, we analyze the applications compilation data and the test execution results. We specify the operation of the tool for category A markers and describe its operation for categories B and C.

Category A markers are strongly related to the analysis of files generated by running tests. The libraries and components unit tests defined above generate significant data. The more functionality the application has, the more it needs test code to evaluate its quality. This quantity affects the readability of application errors. We aim to limit the amount of unnecessary information by selecting errors in our markers. We therefore analyze the result of the execution of the various tests and compile all the errors encountered to make them more readable and more accessible by the developers. We break down the tests into different markers to quickly target the origin of the errors and make their correction easier. Category A markers are therefore lists of the different tests to be performed. If no test fails, we validate the marker; otherwise the list of errors is drawn up.

The operation for markers B and C is different because we search for a code occurrence that triggers a marker. Once this marker has been triggered, we want to validate it by looking for the presence ("must" value) or the absence ("mustnot" value) of code, which validates the marker in the application's source code. In some cases, code in the application forces the presence of other

code, while in another application, it forces the absence of code. A third type of marker exists, the "should" markers, which ensures a possible presence of code. This last value is optional and validates a marker without forcing the presence of code. This makes it possible not to reject or completely invalidate a marker if it does not have the "must" value. As stated above, category B and C markers are defined in JSON files. This choice was made to allow the addition and modification of markers. This is also how we evaluate the different frameworks. We discussed the issue related to the use of frameworks for the cross-platform development of mobile applications. Our tool is independent of these; if a marker is measurable in several frameworks, but in different languages, the marker definition is duplicated in the file and adapted to the framework used. We will see a concrete example on the exploitation of these markers for the Xamarin Framework and the native Android platform. Once the marker is validated, we can list it in a results file that can be consulted by the developers to ensure the quality of the mobile application through these validations. However, if some markers are not validated, they are highlighted to allow developers to fix the problem. The tool we created runs as a result of testing the application's source code and compiling it. It compiles the errors generated by the execution of the tests and analyzes the application's source code. To ensure optimal application quality, the tool can prevent the packaging and delivery of the application if the quality threshold is not reached.

5.2 Implementation

Our tool analyzes the data produced by the compilation, the execution of the tests, and the application's source code. Data are produced whenever the source code is modified by the continuous integration platform. Our tool is launched at the end of this process by the platform. It is executed by a continuous integration platform on a dedicated server. The execution time varies according to the availability of the latter to parallelize the analyses. In addition, the size of the project and the number of classes and dependencies (library) also vary the execution time of the tools.

To describe how the application's source code is analyzed by our tool, we take the example of the permissions in Android. These are needed to enable a mobile application to work properly. For example, permission for a mobile application to access the camera is required when the application wants to use it. We have characterized, through a pattern to search in the code, the use of the camera, and our tool, once the pattern is detected, verifies that the authorization request is correctly implemented in the application's source code.

In another context, we can ensure that a singleton is defined not only in the application, but also in its dependency libraries. This ensures the maintainability of the code through the various libraries integrated in the application.

Secondly, the tool analyzes the mobile application's compilation data. Errors and warnings are sometimes unclear and not highlighted during compilation. Our tools analyze and reformat the compilation errors to highlight them and affiliate them to our markers.

Static code analysis is based on a JSON file (Fig. 2) that describes the tools code requirements. We describe each field below:

—Id and Type correspond to the marker being defined.

—Android and Xamarin are the two target platforms that the tool covers. As we analyze the source code, the latter is dependent on the target platform.

—Inside each platform, we distinguish the "value" search and the "should", "must", and "mustnot" requirements. "Value" represents the pattern that activates the B1 marker. The requirements represent the code whose presence is required (or whose absence is required in the case of a "mustnot" requirement).

Once the tool has finished analyzing all the markers, it describes the need for changes to the application's source code to improve its overall quality.

We gather and group all the errors highlighted by our tool to assess the overall quality of the application.

```
{
"scan": [
{
                "id": "B1",
                "type": "B",
                "label": "camera",
                "android":
 {
                "value": "Camera.open();",
                "should":["ActivityCompat.shouldShowRequestPermissionRationale",
                "Manifest.permission.CAMERA"],
                "must": ["ActivityCompat.shouldShowRequestPermissionRationale",
                "Manifest.permission.CAMERA"]
 },
                "xamarin":
 {
                "value": "CameraSource.Builder",
                "should": ["ActivityCompat.ShouldShowRequestPermissionRationale",
                "Manifest.permission.CAMERA"],
                "must": "Manifest.Permission.CAMERA"
 }
}
}
```

2 - Example: Using the camera in a mobile application that requires permissions.

Developers aim to minimize errors by selecting those that have the greatest impact on the application's quality based on their expertise.

The tools integration is done in the current software production line as well as the continuous integration cycle. We have and use a source code versioning tool to save the different versions of code and to work on the same sources without fear of conflict. The code is divided into branches: each branch represents a code change, often associated with a functionality or a bug fix. These branches are taken from a main development branch. When a development is deemed finished by the developers, the branch is compiled and the analysis tool report on the quality of the current branch is launched. Once the modifications are validated, it can request validation of its source code by integration into the main branch. This request allows other developers to reread the code, which ensures that the development is consistent and understandable. If integration of the development branch into the main branch is allowed, the last tests, compilation, and the analysis tool are performed on the main branch. An intermediate step can be added to retrieve the source code of the main branch on the modification branch. If development takes place in parallel and is added to the main branch after the creation of the modification branch, it is better to manage potential conflicts and errors on the development branch rather than on the main branch. The last branch is the delivery branch, which contains only the application code in the state deliverable to a customer or application stores. The branch does not need to be checked because it corresponds to a copy of the validated main development branch. The delivery branch can also serve as a point of comparison with other deliveries to detect regressions in the functional behavior of the application.

6. RESULTS

Our tool was tested on an application in continuous production, Keysales, choosen among the ones developed by our partner Keyneosoft. We evaluated the source code of the application to provide information on the application quality. Keysales is a mobile application for in-store sales teams, allowing them to create a basket by scanning items in the store, accept payment using a mobile EPT, and print a receipt. The application can also retrieve information from end customers, such as loyalty cards, shopping history, or the current basket on the mobile site. The application comes under the category of products at Keyneosoft because there is a

"white label" version able to adapt to the requirements of the brand that wishes to use it: color schemes, images, and functionalities may be different depending on the brand. These adaptations are currently managed by a configuration file where each parameter activates or deactivates a functionality of the application. The application is therefore constantly evolving because each store may need a new functionality, which is added to the application in such a way that it does not impact the rest of the application. A new parameter is then created in the configuration file. The configuration file depends on the store to which we provide the application, so it is added at the time of compilation of the application for the store. Nevertheless, we aim to validate all applications and also limit the edge effects. For that, we execute our tool on each store, but also on the "white brand" version of the application, which contains all the functionalities in its configuration file. At the end of the sprint, our tool runs for each different compilation and lists the errors it has detected based on the different markers tested.

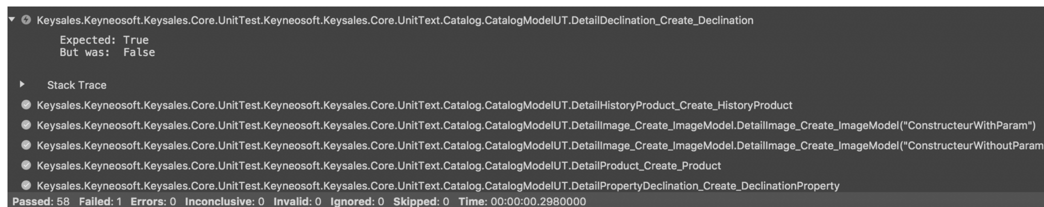
6.1 List of uploaded data

The error list is stored in a file that developers can search to ensure that the application correctly passes all the quality markers.

Errors are categorized for ease of reading.

The first ones are the compilation errors returned by the platform compiler.

We then have the unit and integration testing results for category A, followed by the list of category B and C markers that have not been validated.



3 - Compile error list.

```
Project dependencie fount at : ..\packages\Xamarin.Build.Download.0.4.7\build\Xamarin.Build.Download.props
…
Project dependencie fount at : ..\packages\Xamarin.GooglePlayServices.Vision.60.1142.0\build\MonoAndroid80\Xamarin.GooglePlayServices.Vision.targets

Reference dependencie fount at : "HockeySDK, Version=5.0.6585.25315, Culture=neutral, processorArchitecture=MSIL">
…
Reference dependencie fount at : "Keyneosoft.Adyen.Plugin">


Microsoft (R) Build Engine version 15.4.0.0 (xplat-master/67e8006d Fri Feb 23 21:24:28 EST 2018) for Mono
Build started 05/04/2018 09:31:15.
…
Build succeeded.

TEST File :  0802_RGE_XAM_RougeGorge/RougeGorge/RougeGorge/App/View/Scan/ScanView.cs
FOR :             cameraSource = new CameraSource.Builder(this.Context, barcodeDetector)
VALIDATE File :  0802_RGE_XAM_RougeGorge/RougeGorge/RougeGorge/App/View/Scan/ScanView.cs contains : Manifest.Permission.CAMERA

  Conflit détecté entre 'Reference:/usr/local/share/dotnet/sdk/NuGetFallbackFolder/netstandard.library/2.0.1/build/netstandard2.0/ref/System.Reflecti
packages/system.reflection.extensions/4.0.1/ref/netstandard1.0/System.Reflection.Extensions.dll'.  'Reference:/usr/local/share/dotnet/sdk/NuGetFallba
System.Reflection.Extensions.dll' choisi, car AssemblyVersion '4.0.1.0' est supérieur à '4.0.0.0'.
…
  Conflit détecté entre 'Reference:/usr/local/share/dotnet/sdk/NuGetFallbackFolder/netstandard.library/2.0.1/build/netstandard2.0/ref/System.Xml.XDoc
system.xml.xdocument/4.0.11/ref/netstandard1.3/System.Xml.XDocument.dll'.  'Reference:/usr/local/share/dotnet/sdk/NuGetFallbackFolder/netstandard.lib
choisi, car AssemblyVersion '4.0.11.0' est supérieur à '4.0.10.0'.

    89 Warning(s)

    0 Error(s)
```

4 - Code Analysis error list

In these figures (Fig. 3 and Fig. 4), we distinguish the list of errors that can be recovered and warnings that have a lesser impact on quality. Currently, this list is for developers as it is very

technical. Nevertheless, it is possible to determine which quality markers are invalid and which lines of code are associated with the errors. Category A errors are often related to the modification of the existing method, but not that of the associated test. Developers must first ensure that the development's source code remains in line with the technical and functional expectations before modifying the test code. If this is the case, they can modify the test code to adapt it to the source code modification. This error is very frequent on large applications because even if numerous tests are run, the time granted to the development and the edition of the test code are rarely respected. Category B and C errors are directly related to the markers, so the errors presented are traceable in the source code. For example: "error B1 internet permission mainView.java". This allows developers to modify the source code of the application to satisfy the needs of the invalid marker. Any modification of the source code or the test code is subjected to compilation and the analysis tool. In our case, the complete execution of the continuous integration platform does not exceed two minutes. Repetition of this execution does not pose a performance problem. When an error is corrected, the tool checks that the correction is valid and that it does not lead to a regression or an edge effect in the application's functionalities. The error lists are compiled and archived to evaluate the tool's performance and the relevance of the errors reported.

The time saving is difficult to quantify because the errors reported are very different in nature. Some of them are now avoided during local compilation by IDEs (native Android) or specific plugins while others are compensated by developers' experience. But, from our experience, we faced different issues that takes a sometimes a full day work to identify in some project not using the tool, whereas they could have been highlighted in a few minutes thanks to it.

6.2 Use of feedback

In this section, we describe how to proceed once the list of errors has been reported. As mentioned above, the list of errors is sorted by category. The first list corresponds to errors related to the compilation and execution of unit tests, which are the first errors to be processed. They impact the mobile quality the most because they reflect a problem of the very functioning of the application. Subsequently, category B and C errors can be processed because they are mainly defined to compensate for any problem. Code correction is like editing or adding functionality. Changes and corrections are made to test and evaluate the new code. The changes are then sent to the branch causing the error to validate the incorporation. This incorporation then leads to a delivery. Most of the markers that are present in the tool are the result of feedback from experts at Keyneosoft. From there, the deliveries made to the customer (in our case for an application used by multiple stores) show an increase in quality in the sense that the customer's negative point returns decrease. In addition, evolutions are often faster because errors are more easily identified.

7. CONCLUSION AND PERSPECTIVES

We used our tool on an application currently in production. We choose one from our partner, Keynesoft : the KeySales application. This product is an application for various retail stores worldwide. It proposes to facilitate the mobility of an in-store sales team while adapting to the requirements of the store in question. The functionalities of the application vary and must coexist. The application is a perfect testing ground for our tool because it is subjected to numerous code modifications and therefore potential impacts on quality. We used our tool on several developments to evaluate the quantity of error returned. We recorded a decrease of errors in the entire application because the errors impacting quality were highlighted. Over a period of four months, the tool was launched at the end of each two-week sprint on the main development branch. The tool revealed 10 category B errors, and four category A markers were invalidated, resulting to code corrections. The tool also analyzed the compilation and execution results of nearly 50 tests. Once the application is delivered, we would like to be able to trace the information on the use of application in production. Some of our criteria validated during development can have a different behavior under production conditions, for example, the number of users can affect

the quality of an application. We would like to implement trackers in applications to validate our markers continuously over time, even after development, by adding generated code inside the source code. The generated code presents a potential risk for the quality of mobile applications; therefore, we are considering this step as a potential development of our work.

Another potential development concerns the usability and ergonomics of the tool. As the tool is a prototype, its use is limited. We said above that the quality markers defined in files, although easily modifiable, can be modified by only a few people. A way to add or remove file markers could be considered.

It would also be interesting to be able to disable markers as some markers may not interest developers, despite their relevance.

Finally, a potential evolution focuses on the prioritization of detected invalid markers. In some cases, it may be preferable to correct some failed quality points before others. Although our tool highlights errors related to application quality, and even if quality concerns everyone, real investment in quality is limited because it is not directly profitable.

REFERENCES

Bisiaux, R., Desertot, M., Lecomte, S., Perchat, J., and Petit, D. 2017. Improving quality on native and cross-platform mobile application. In *Mobility: The Seventh International Conference on Mobile Services, Resources, and Users.*

Cai, X.,Lyu, M.R.,Wong, K.-F., and Ko, R. 2000. Component-based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes. In *Proceedings of the Seventh Asia-Pacific Software Engineering Conference, APSEC 00. IEEE Computer Society, Washington, DC, USA, pp. 372.*

Coutaz, J.,Crowley, J.L.,Dobson, S., and Garlan, D., 2005. Context is key. Commun. ACM 48, 4953. https://doi.org/10.1145/1047671.1047703

Garofalakis, J., Stefani, A., Stefanis, V., and Xenos, M., 2007. Quality Attributes of Consumer-Based m-Commerce Systems. In *Proceedings of the International Conference on e-Business, ICE-B is part of ICETE - The International Joint Conference on e-Business and Telecommunications, Volume: Barcelona, Spain, July 28-31, 2007pp. 130136.*

Dehlinger, J., and Dixon, J., 2011. Mobile Application Software Engineering: Challenges and Research Directions. In *Workshop on Mobile Software Engineering*

Dickson, J., 2013. Xamarin Mobile Development. Technical Library.

Duvall, P.M., Stephen, M., and Glover, A., 2007. Continuous Integration: Improving Software Quality and Reducing Risk In *Addison-Wesley Signature Series (Fowler). Addison-Wesley Professional. isbn=978-0-321-33638-5*

Franke, D., and Weise, C., 2011. Providing a software quality framework for testing of mobile applications. In *Fourth IEEE International Conference on Software Testing, Verification and Validation. IEEE, pp. 431434.*

Hallsteinsen, S., Hinchey, M., Park, S., and Schmid, K., 2008. Dynamic software product lines. In *Computer Volume 41 issue 4, IEEE Computer Society Press, 9395.*

Joorabchi, M.E., Mesbah, A., and Kruchten, P., 2013. Real challenges in mobile app development. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 1524.*

Perchat, J., Desertot, M., and Lecomte, S., 2014. Common framework: A hybrid approach to integrate cross-platform components in mobile application. In *Journal of Computer Science, Volume 10, p2165-2181, doi=10.3844/jcssp.2014.2165.2181*

Popovici, D., Desertot, M., Lecomte, S., and Peon, N., 2011. Context-Aware Transportation Services (CATS) Framework for Mobile Environments. In *International Journal of Next-Generation Computing, Volume 2.*

Popovici, D., Desertot, M., Lecomte, S., and Delot, T., 2012. A framework for mobile and context-aware applications applied to vehicular social networks. In *Soc. Netw. Anal. Min. (2013) 3: 329. https://doi.org/10.1007/s13278-012-0073-9, Springer Vienna.*

Pree, W., 1995. Design patterns for object-oriented software development. In *ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, isbn=0-201-42294-8*

Schwaber, K., and Sutherland, J., 2017. The scrum guide. https://www.scrum.org/resources/scrum-guide

Syromiatnikov, A., and Weyns, D.. A Journey through the Land of Model-View-Design Patterns. In *Proceedings - Working IEEE/IFIP Conference on Software Architecture 2014, WICSA 2014. pp. 2130. https://doi.org/10.1109/WICSA.2014.13*

Team, C.P., 2006. CMMI for Development, Version 1.2. In *Carnegie Mellon University/Software Engineering Institute. Technical report CMU/SEI-2006-TR-008.*

Zahra, S., Khalid, A., and Javed, A., 2013. An Efficient and Effective New Generation Objective Quality Model for Mobile Applications. In *International Journal of Modern Education and Computer Science 5, 3642. https://doi.org/10.5815/ijmecs.2013.04.05*

Zubrow, D., 2004. Software quality requirements and evaluation, the ISO 25000 series. In *Software Engineering Institute, Carnegie Mellon.*

Rudy Bisiaux is a PhD student at LAMIH-Polytechnic University of Hauts-de-France since 2015. He obtained his Master degree in Computer Science in this same university. his thesis is funded by Keyneosoft, a company providing digital and omni-channel solutions dedicated to new purchasing paths. He is an expert on native or cross-platform mobile technologies.

Mikael Desertot is assistant professor at LAMIH-Polytechnic University of Hauts-de-France since 2008. He defended his PhD in computer science in 2007 after working 3 years in collaboration with Bull SA on dynamic application component containers for Java. He then spent one year and a half working in Canada for Oracle Corp. Currently, his research area focuses on dynamic software architecture and reconfiguration, relying on service oriented architectures, and targeting mobile application development and quality, for transportation services and human assistance.

Sylvain Lecomte He is Full Professor at LAMIH-Polytechnic University of Hauts-de-France. He has obtained a HDR in Computer Science at the University of Valenciennes (Title: Conception and adaptation of technical services dedicated to ambient computing , 2005) and a PhD in Computer Science at the University of Lille 1 (Title: COST-STIC : Smart Card embedded into transactional services and transactional services embedded into smart Card, 1998). Relevant Work Experience: Strong background in Context aware computing, Mobile services and Component model

Dorian Petit is an associate professor in the Department of computer science at LAMIH (UMR CNRS 8201)-Polytechnic University of Hauts-de-France. He received his Ph.D. in computer science from University of Valenciennes (title: Generating trusted software components from formal specifications). His research is focused on safety of software in transportation systems..