An UI Layout Files Analyzer for Test Data Generation

Paul POCATILU, Felician ALECU
Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies, Romania
ppaul@ase.ro, felician.alecu@ie.ase.ro

Prevention actions (trainings, audits) and inspections (tests, validations, code reviews) are the crucial factors in achieving a high quality level for any software application simply because low investments in this area are leading to significant expenses in terms of corrective actions needed for defect fixing. Mobile applications testing involves the use of various tools and scenarios. An important process is represented by test data generation. This paper proposes a test data generator (TDG) system for mobile applications using several sources for test data and it focuses on the UI layout files analyzer module. The proposed architecture aims to reduce time-to-market for mobile applications. The focus is on test data generators based on the source code, user interface layout files (using markup languages like XML or XAML) and application specifications. In order to assure a common interface for test data generators, an XML or JSON-based language called Data Specification Language (DSL) is proposed.
Keywords*: Mobile Applications, Software Testing, Test Data Generators, Software Quality, Time-to-Market, UI Layout*

1 Introduction

Software testing is an important phase within the software development cycle. It is also an expensive phase counting about 40% of the total cost associated with software development [1], [2].

According to ISO9000 standard [13], the quality associated to a deliverable is seen as being "the degree to which a set of inherent characteristics fulfill requirements". If the requirements are not (completely) met, severe consequences may follow, like intensive rework or even the risk of losing the business.

From an economic point of view, quality measures are indicating the application actual metrics (in terms of cost, time and scope) that can be easily compared with the planned ones. To fulfill the requirements is crucial but the way in which these requirements are met in terms of time, cost and application scope is also very important because an application may be delivered in time but with a lot extra work and/or by using lighter inspections, so the project may not be so successful due to the reduced profits and very possible future implementation costs.

When discussing about the quality of a software application, usually there are two types of costs assigned [14]:

- poor quality costs – defects that must be repaired by corrective actions;
- good quality costs – code inspections and prevention activities, like planning, training and auditing.

As illustrated into the Figure 1, any application can reach a great quality level at a reasonable cost by positioning itself just above the optimal point indicated by the vertical line.

Testing methodologies include white-box and black-box approaches [3], [4]. White-box testing considers the internal structure of programs. In this case, test data generation uses several coverage criteria (path, statement, branch etc.). Black-box testing does not depend on the internal structure of programs. Test data is generated based on program specifications.

One area of software testing that can be automated is test data generation. Test data generation based on coverage can use random or optimized inputs, engaging advanced techniques [5], [6].

DOI: 10.12948/issn14531305/18.2.2014.06

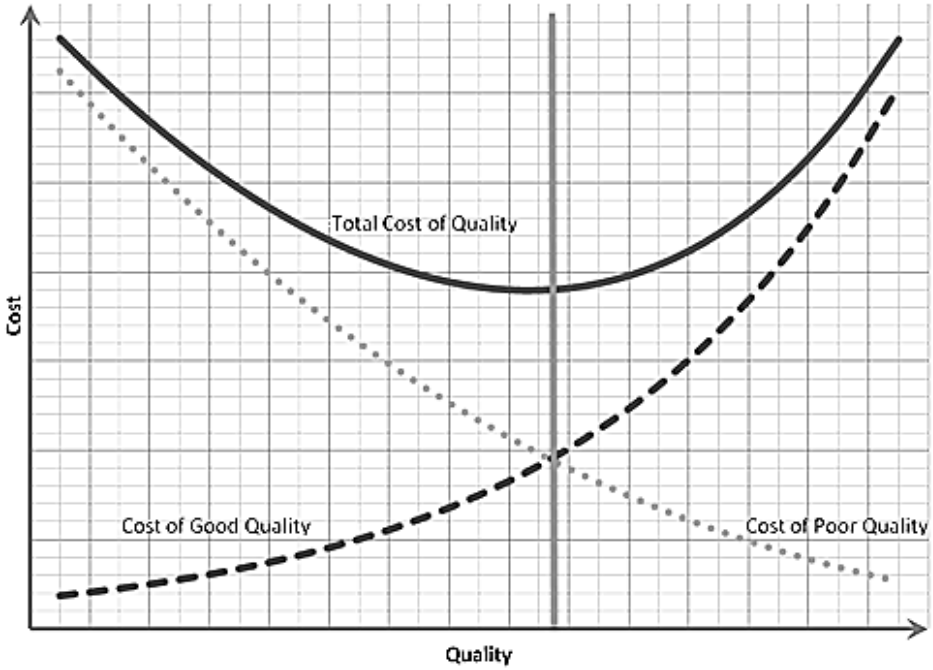

Fig. 1. Total Cost of Quality

There is a continuous uptrend of mobile application development. Compared with other software solutions, time-to-market for mobile applications tends to be shorter. A mobile application can be used on different versions of the target platform, on different hardware configurations (screen size, processor architecture, processor speed, memory capacity etc.), different types of networks (Wi-Fi, mobile) etc. In this respect, mobile applications testing is more challenging than other type of application testing [7], [8], [9]. Every improvement of testing process for mobile applications helps achieving high quality software, reducing the development costs and the time-to-market.

We propose a complex system for automated test data generation for mobile applications in order to reduce the required time for testing while keeping the same level of quality. This integrates our previous work that includes researches related to test data generators based on source code as in [6] and [10] where we proposed a test data generator using genetic algorithms and a framework for test data generators analysis. The system combines several inputs for test data generator in order to achieve an optimized set of inputs that assure the best coverage of code and the input domain.

The paper is organized as follows: Section 2 describes the proposed framework for test data generators for mobile applications and the Data Specification Language (DSL). Section 3 presents the proposed components of the test data generator that are based on source code, user interface layout files and specification. Section 4 presents the same example implemented for two mobile platforms (Android and Windows Phone) in order to highlight the UI layout files challenges. Section 5 is dedicated to the discussion related to presented work. The paper ends with conclusions and future work.

2. Test Data Generators

The TDG is a complex system that include several subsystems that are linked together by inputs and outputs. The proposed system consists of the following components:

- Source code analyzer;
- User interface files analyzer;
- Specification analyzer;
- Test data generator.

The input of the system consists of the source files, user interface layout files and specifications. The system components are depicted in Figure 2.

DOI: 10.12948/issn14531305/18.2.2014.06

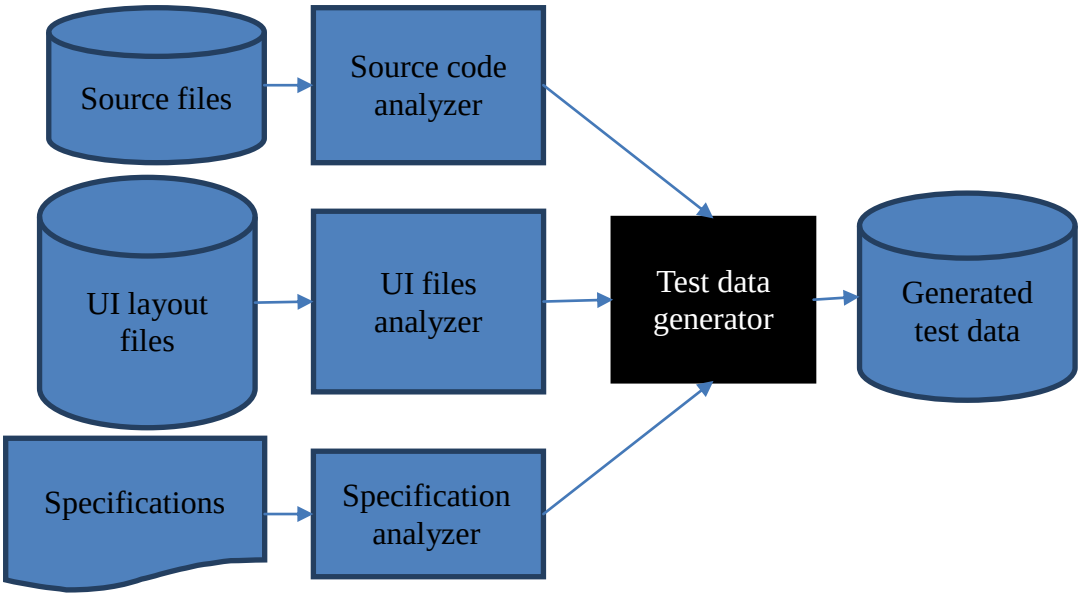

Fig. 2. Test data generator inputs and outputs

One type of input for data generator consists of one or more files based on DSL (Data specification Language). This is an XML-based language used to describe data type, data length, type of generation (random or based on list of values), number of occurrences, data boundaries etc. An example of a DSL content is presented in Listing 1.

Listing 1. Example of DSL content

XML

```
<dataset>
	<field>
		<type>string</type>
		<length>30</length >
	</field>
	<field>
		<type>number</type>
		<length>int</length >
		<values>
			<start>0</start>
			<end>1000</end>
		</values>
	</field>
	<field>
		<type>number</type>
		<length>byte</length >
		<values>
			<value>0</value>
			<value>1</value>
		</values>
	</field>
</dataset>
```

JSON

```
{
  "dataset": {
    "field": [ {
        "type": "string",
        "length": "30"
      },
     {
        "type": "number",
        "length": "int",
        "values": {
          "start": "0",
          "end": "1000"
        }},
     {
        "type": "number",
        "length": "byte",
        "values": {
          "value": [
            "0",
            "1"
          ] } } ] }
}
```

The framework is flexible enough to accept any other components that will provide data specification files based on DSL.

The test data generator is further detailed in Figure 3. There are at least two types of inputs so there will be two distinct data generators, each one using a different way to produce the expected test data.

The instrumented source files are the output of the source files analyzer. These will allow to record the code coverage by executing it.

DOI: 10.12948/issn14531305/18.2.2014.06

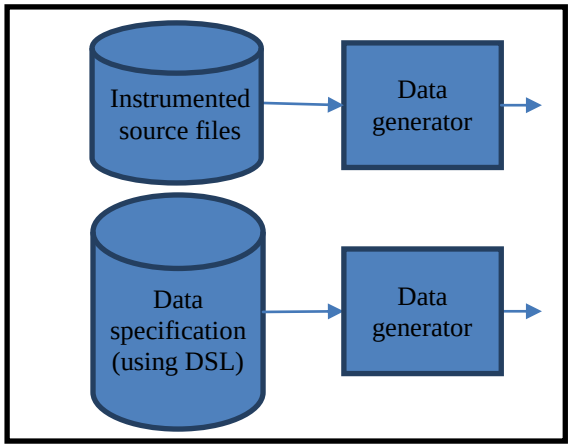

Fig. 3. Test data generator components

The outputs of UI files analyzer and specification analyzer are DSL files that represent the input of data generator. This will parse the DSL files and will produce test data for the software under test (SUT).

3 TDG Analyzers

3.1 Source Code Instrumentation

This module will parse the source code and will instrument the files in order to record the code execution. Structural testing is subject to numerous papers, including [11] and [12]. Test data generators are based on the associated control flow graph (CFG), control flow diagram (CFD) or program tree.

After the source code is instrumented, it is used as input by the data generator. This will generate test data either randomly, or using evolutionary algorithms. The SUT will be executed using the generated data in order to achieve the targeted coverage criteria, depending on the required level: statements, branches, paths, blocks, data flows or functions [6].

The source code is modified by adding simple calls to output functions or to more complex functions. While SUT is running, the calls will record the executed functions to calculate the coverage with the current data set.

The source code analyzer could also generate the associated CFG, CFD or the program tree that can be used as inputs for other types of data generators.

3.2 User Interface Layout Files

Several mobile application development include, as other platforms, the possibility to declare the user interface layouts using separate files based on markup languages. For example, Android developers use XML files to declare the interface, iOS developers create storyboards while the Windows Phone developers rely on XAML files.

The analyzer will parse the UI layout file and, for each control, will determine the associated properties. Based on the control type and the associated attributes the analyzer will generate test data specification using DSL format.

For example, Table 1 presents an example for the basic input text control as defined by different UI layout markup languages. Several sources such as [15], [16], [17], [18] and [19] present the content and structure of Android, iOS and Windows Phone UI layout files.

Table 1. Example of input text declaration

UI layout file	Input text tag	Input type
Android (xml)	EditText	inputType= "number"
iOS (storyboard)	textField	keyboardType="numberPad"
Windows Phone (xaml)	TextBox	InputScope="Number"

If the input type is specified, this will help in determining the data type for that input text control. Table 1 shows that the exemplified fields requires numeric input and so the test data domain will be narrowed to this type.

Other controls may have two or more states or they can use a list of predefined values. All of these can be used to describe the data that will be generated for the component under test.

3.3 Specifications

The specifications represent one of the most important source for the test data generation.

DOI: 10.12948/issn14531305/18.2.2014.06

The level of details (user and system specification), the representation and the storage medium for specifications may vary from one team to another. This high level of variation generates some challenges for generalization. In this respect, the analyzer will require a preprocessing phase that will unify the input for the specification analyzer.

As stated in [2], the requirements are of two kinds, functional (related directly to the application) and nonfunctional (usability, performance, security etc.). Test data generator should focus on both types of requirements.

Each component under test (module, class, screen etc.) will require the associated specification files. This could include inputs, outputs, source, destination, actions, pre-conditions and post-conditions etc.

4 UI Layout Files Analysis

In order to implement the UI layout file analyzer, the same example will be used for two platforms: Android and Windows Phone. The example consists of a screen used to collect user data such as first and last name and age.

The UI layout analyzer should detect three text fields used for data entry. The third field (*age*) should contain only numbers.

An excerpt from the UI layout file for the Android application is presented in Listing 2.

For this example the targeted controls are **EditText** type. For these controls the **android:inputType** attribute is located in order to determine the input data type.

Another attribute for the **EditText** control is **android:maxLength** that gives information related to the maximum text size that will be generated for this field.

Listing 2. Excerpt from the XML layout file for Android application

```
<TextView
  android:id="@+id/textView1"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="First Name" />

<EditText
 android:id="@+id/editTextFirstName"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_alignLeft=
    "@+id/textView1"
 android:layout_below="@+id/textView1"
 android:layout_marginTop="16dp"
 android:ems="10"
 android:inputType="textCapWords" />

<TextView
 android:id="@+id/textView2"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_alignLeft=
    "@+id/editTextFirstName"
 android:layout_below=
    "@+id/editTextFirstName"
 android:text="Last Name" />

<EditText
 android:id="@+id/editTextLastName"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_alignLeft=
    "@+id/textView2"
 android:layout_below=
    "@+id/textView2"
 android:layout_marginTop="16dp"
 android:ems="10"
 android:inputType="textCapWords" />
<TextView
 android:id="@+id/textView3"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_alignLeft=
    "@+id/editTextLastName"
 android:layout_below=
    "@+id/editTextLastName"
 android:text="Age" />

<EditText
 android:id="@+id/editTextAge"
 android:layout_width="50dp"
 android:layout_height="wrap_content"
 android:layout_alignLeft=
     "@+id/textView3"
 android:layout_below="@+id/textView3"
 android:layout_marginTop="16dp"
 android:ems="10"
 android:inputType="number" />

<Button
 android:id="@+id/btnOK"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_alignLeft=
     "@+id/editTextAge"
 android:layout_alignParentBottom="true"
 android:layout_marginBottom="23dp"
 android:text="OK" />

<Button
 android:id="@+id/btnCancel"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_alignBaseline=
     "@+id/btnOK"
 android:layout_alignBottom=
```

DOI: 10.12948/issn14531305/18.2.2014.06

```
                "@+id/btnOK"
            android:layout_alignParentRight="true"
            android:layout_marginRight="39dp"
            android:text="Cancel" />
```

The **textCapWords** value for **android:inputType** attribute is treated as a simple string.
Figure 4 represents the screenshot of the Android application screen associated to the layout presented in Listing 2.

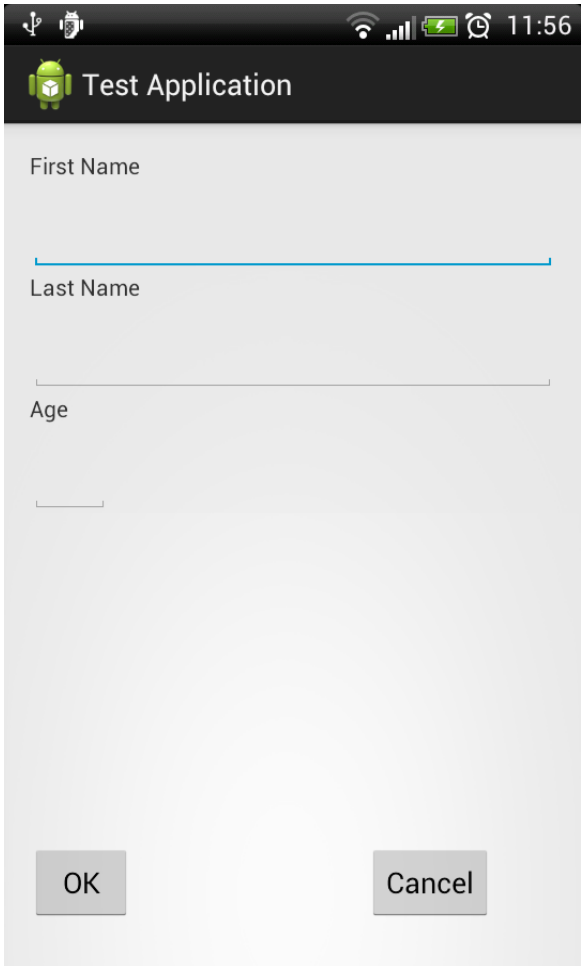

Fig. 4. Example of a test screen from the Android application

The same functionality was implemented in a Windows Phone application. Listing 3 is an excerpt from the associated XAML file. The UI layout analyzer module for Windows Phone UI layout files will parse the content looking (in this example) for **TextBox** controls. Within **TextBox** tags the **InputScope** attribute is located in order to determine the input type. The maximum number of characters is controlled by **MaxLength** attribute. Similarly to Android layout files, the **PersonalSurname** and **PersonalGivenName** values for **InputScope** attribute will be treated as strings by the DSL generator.

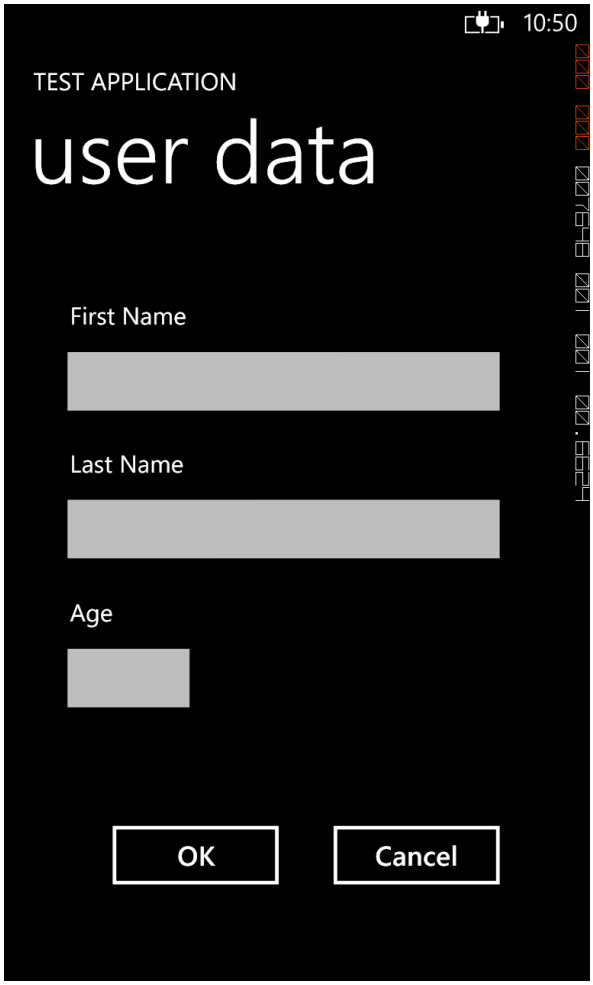

Fig. 5. Example of a test screen from the Windows Phone application

Figure 5 depicts the same application screen but for Windows Phone platform. The associated UI layout file is presented in Listing 3.
The UI layout file analyzer could be extended also for other input controls such as lists or combo-boxes and also for buttons in order to support testing automation. This represents a future direction for automation of user interaction testing for mobile applications.

DOI: 10.12948/issn14531305/18.2.2014.06

Listing 3. Excerpt from the XAML layout file for Windows Phone application

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBox Height="72" HorizontalAlignment="Left" Margin="28,80,0,0"
x:Name="textBoxFirstName" VerticalAlignment="Top" Width="378" InputScope="
PersonalGivenName" />
    <TextBox Height="72" HorizontalAlignment="Left" Margin="28,201,0,0"
x:Name="textBoxLastName" VerticalAlignment="Top" Width="378" InputScope="
PersonalSurname" />
    <TextBox Height="72" HorizontalAlignment="Left" Margin="28,323,0,0"
x:Name="textBoxAge" VerticalAlignment="Top" Width="124" InputScope="Number" />
    <TextBlock HorizontalAlignment="Left" Margin="42,48,0,0" TextWrapping="Wrap"
Text="First Name" VerticalAlignment="Top"/>
    <TextBlock HorizontalAlignment="Left" Margin="42,169,0,0" TextWrapping="Wrap"
Text="Last Name" VerticalAlignment="Top"/>
    <TextBlock HorizontalAlignment="Left" Margin="42,291,0,0" TextWrapping="Wrap"
Text="Age" VerticalAlignment="Top"/>
    <Button x:Name="buttonOK" Content="OK" HorizontalAlignment="Left"
Margin="65,468,0,0" VerticalAlignment="Top" Width="160"/>
    <Button x:Name="buttonCancel" Content="Cancel" HorizontalAlignment="Left"
Margin="246,468,0,0" VerticalAlignment="Top" Width="160"/>
</Grid>
```

The UI layout analyzer will parse each layout file. The parser should be implemented for each mobile platform under test in order to correctly detect the user controls.
The resulting DSL file (XML-based) is presented in Listing 4.

Listing 4. Example of DSL generated content

```
<dataset>
   <field>
       <type>string</type>
       <length>MAX_STRING</length>
   </field>
   <field>
       <type>string</type>
       <length>MAX_STRING</length>
   </field>
   <field>
       <type>number</type>
       <length>MAX_NUMBER</length>
   </field>
</dataset>
```

If the data length cannot be deduced from the UI layout files, several constants representing the maximum limit will be used (MAX_STRING, MAX_NUMBER etc.). If the data length is identified within the controls attributes, these will be used as limits. For lists, the contained values will be used for input.
Before the DSL file will be used by the test data generator, it could be correlated with the specification and adjusted in order to provide the best description for test data.

5 Discussions

The test data generator designed for mobile applications include several analyzers that generate data specification files (based on DSL markup language) or instruments the source code. Data generators are implemented differently based on their inputs. Table 2 presents the main characteristics identified for data generators depending on their functionality.
The analyzers for source code and user interface files require total or partial access to the application source files. The UI layout files are part of the application project and have to be parsed by the analyzer. The analyzer results can be combined to obtain test data specifications. The specification analyzer does not require access to any of the source files.
The source code analyzer and the UI layout files analyzer require knowledge about the internal structure of the application. These modules are closely related to the developer. The specification analyzer is independent of the application implementation and it is closed to the user or customer.
The code required to generate test data based on source files has a high complexity. TDGs are dependent on the programming language. Firstly, a source files has to be instrumented. Then, data generator will run SUT in order to record the code coverage. For UI layout files the parser is language dependent and it is

DOI: 10.12948/issn14531305/18.2.2014.06

possible to be implemented for several markup languages. The specification analyzer is less complex, but it requires a preprocessing phase, that could include manual work, in order to receive the desired input.

The main advantages of the proposed solution are:

- a broader source for test data generation;
- it targets structural and functional testing approaches;
- flexible design, that allows the development and use of other components;
- uses standard input and output files in order to allow a common interface.

The UI layout files analyzer is a useful tool for test data generation. It has a medium complexity, but could be very helpful in generating test data, especially when the specifications are missing or are incomplete.

Table 2. Comparison of test data generators flows

Analyzer	Source code access	Knowledge about the application implementation	Complexity
Source code	Yes	Required	High
User interface layout files	Partial	Partially required	Medium
Specifications	No	Not required	Low

The proposed solution has components with a high level of complexity and it has to be adapted for each targeted platform.

6 Conclusion and Future Work

Reaching a high quality level for a software application can only be done by massively investing in prevention. This is why it is very important to position ourselves just above the optimal value of the TCQ (Total Cost of Quality) where the quality costs are minimal and the quality level is high enough.

Due to the exponential grow of the TCQ shape, a very high quality level turns to be somehow uneconomical since the investments needed are much higher than the achievements in terms of quality.

Test data is very important for the success of any application. Test data is required at several levels that are closed to the developer (source code) or to the users/customers of the developed application. There is no single solution for test data generation and these available options have to be used together in order to obtain the best results. The proposed system is based on several inputs for test data generation and it could be extended for other types of input. Each input will be used by a dedicated analyzer that will generate outputs for data generators. Finally, data generators will produce test data for the software under test.

On the short term, the future work will include the development of the UI layout files parser for mobile applications together with the associated data generator, and a detailed description of DSL language. On the long term we intend to implement the proposed solution in order to be used in a real environment.

Acknowledgment

Parts of this research have been published in the Proceedings of the 13th International Conference on Informatics in Economy, IE 2014 [20].

References

[1] S. Pressman, Software Engineering: A Practitioner's Approach. 7th ed., New York: McGraw-Hill, 2009.

[2] I. Sommerville, Software Engineering. 9th ed., Boston: Addison-Wesley, 2011.

[3] M. Roper, Software Testing, McGraw-Hill Book, 1994.

[4] G. J. Meyers, The Art of Software Testing, Second Edition, New Jersey: John Wiley & Sons, 2004.

[5] M. Harman, F. Islam, T. Xie and S.

Wappler, "Automated Test Data Generation for Aspect-Oriented Programs," in *Proc. of AOSD'09*, Charlottesville, 2009.

[6] P. Pocatilu and I. Ivan, "A Genetic Algorithm-based System for Automatic Control of Test Data Generation," *Studies in Informatics and Control,* vol. 22, no. 2, pp. 219-226, 2013.

[7] M. Kumar and M. Chauhan, "Best Practices in Mobile Application Testing (White Paper)," Infosys, Bangalore, 2013.

[8] R. Selvam and V. Karthikeyani, "Mobile Software Testing – Automated Test Case Design Strategies," *International Journal on Computer Science and Engineering (IJCSE),* vol. 3, no. 4, pp. 1450-1461, 2011.

[9] P. Pocatilu, "Testing Java ME Applications," *Informatica Economică,* vol. 12, no. 47, pp. 147-150, 2008.

[10] P. Pocatilu, "A Framework for Test Data Generators Analysis," *Economic Computation and Economic Cybernetics Studies and Research,* vol. 47, no. 3, pp. 185-198, 2013.

[11] S. Jiang, Y. Zhang and D. Yi, "Test Data Generation Approach for Basis Path Coverage," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 3, pp. 1-7, 2012.

[12] S. Varshney and M. Mehrotra, "Search based Software Test Data Generation for Structural Testing: A Perspective," *ACM SIGSOFT Software Engineering Notes,* vol. 38, no. 4, pp. 1-6, 2013.

[13] Project Management Institute, *A Guide to the Project Management Body of Knowledge: PMBOK Guide,* 5th edition, Project Management Institute, 2013

[14] International Standards Organization. 2008. ISO 9000:2008. Quality Management Systems – Fundamentals and Vocabulary. Geneva, Switzerland: ISO.

[15] R. Meier, *Professional Android 4 Application Development*, Wiley, 2012

[16] S. Komatineni and D. MacLean, *Pro Android 4*, Apress, 2012

[17] P. Pocatilu, *Programarea dispozitivelor mobile*, ASE Publishing House, 2012

[18] T. Szostak, *Windows Phone 8 Application Development Essentials*, Packt Publishing, 2013

[19] M. Neuburg, *iOS 7 Programming Fundamentals*, O'Reilly Media, 2013

[20] P. Pocatilu, F. Alecu and S. Capisizu, A Test Data Generator for Mobile Applications, *Proceedings of the IE 2014 International Conference*, Bucharest, Romania, May 15-18, 2014, ISSN 2284-7472, pp. 116-121

Paul POCATILU graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 1998. He achieved the PhD in Economics in 2003 with thesis on Software Testing Cost Assessment Models. He has published as author and co-author over 45 articles in journals and over 40 articles on national and international conferences. He is author and co-author of 10 books, (Mobile Devices Programming and Software Testing Costs are two of them). He is professor at the Department of Economic Informatics and Cybernetics within the Bucharest University of Economic Studies, Bucharest. He teaches courses, seminars and laboratories on Mobile Devices Programming, Economic Informatics, Computer Programming and Project Quality Management to graduate and postgraduate students. His current research areas are software testing, software quality, project management, and mobile application development.

DOI: 10.12948/issn14531305/18.2.2014.06

Felician ALECU has graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 2000 and he holds a PhD diploma in Economics from 2006. Currently he is lecturer of Economic Informatics within the Department of Economic Informatics at Faculty of Cybernetics, Statistics and Economic Informatics from the Academy of Economic Studies. He is the author of several articles in the field of parallel computers, grid computing and distributed processing. He holds a Project Management Professional (PMP) certification from the Project Management Institute (PMI), and he is member of the Romanian chapter of PMI.

DOI: 10.12948/issn14531305/18.2.2014.06