Preprocessor Based Approach for Cross-Platform Development with Qt Quick Components

Ilya Paramonov, Andrew Vasilev,
Denis Laure, Nikita Kozhemyakin
Yaroslavl State University
Yaroslavl, Russia
Ilya.Paramonov@fruct.org
{vamonster, den.a.laure, enginegl.ec}@gmail.com

Abstract

In the paper we analyze the obstacles of cross-platform mobile development involving Qt Quick Components library. We provide the classification of the most common issues of such a development and propose resolution for them with the use of the specially developed tool—QML preprocessor.

Our approach is proven to be successful in development of two mobile applications supporting both Harmattan and Symbian platforms.

Index Terms: QML, Qt Quick Components, preprocessor.

I. Introduction

Qt platform keeps running away from conventional widgets that were with us for so many years. Declarative interfaces and focus on touch-based interfaces lead to introduction of Qt Quick platform, that supports this behavior. It became primary successful on mobile devices: tablets and smartphones, but fully supported on personal computers.

Qt Quick provides all basic primitives, that are necessary to build user interface from scratch, but there are no any conventional widgets present, therefore it requires more effort to create applications which provide familiar elements like buttons and lists, because developer needs to create required widgets first. It definitely could be useful if project designer wants to express functionality in a non-standard way, but it turns out to be a problem if an application should provide some generic functionality like settings window.

Nokia initiated two projects with the target to create a set of QML elements which could support already known interaction models, including lists, selection dialogs, calendars and other. First QML element library is called Colibri [1], it was created within a research project. Widgets implemented there are based solely on Qt Quick platform and therefore platform independent. Unfortunately, these elements do not provide the satisfactory level of productivity and stability to create even basic applications using it. The development of the library was stopped in the January 2011, but all components are available on the project site.

The other project is called Qt Quick Components [2]. Nowadays it is the mainstream library developed by Nokia labs. The project target is to provide items for several platforms: Symbian, MeeGo Harmattan, and desktop. Having the library targeted at several platforms, it seems naturally to utilize it for cross-platform development to extend the potential availability of the application while effectively reusing the same code.

On the first look, the components for different platforms provide similar API, but actually they only share basic architecture decisions and have separate implementations for each of

the target platforms. Multiple incompatibilities in API and behavior of the same components make really hard cross-platform development using this library. The obvious solution is to use separate QML elements for each of the target platform. Unfortunately, this approach leads to code duplication, as a developer should implement one idea at least two times using similar APIs, which potentially induces errors in code and leads to maintainability issues.

It turns out that cross-platform development involving Qt Components requires more technological solution, which avoids code duplication and follows "Do not Repeat Yourself" (DRY) principle [3]. In this paper we describe such a solution. It is based on isolation of differences in Qt Components API and behavior inside separate components and the use of specially developed tool (QML preprocessor) to allow sharing of the same code base in Qt Components based applications targeted at all available platforms.

The paper is structured as following. In section II we analyze the issues of cross-platform development due to the differences in implementation of Qt Components. As a result of the analysis, we provide a classification of these issues. Section III is devoted to description of the QML preprocessor, which was developed by us to cope with these issues. Our solutions on how to resolve the cross-platform development issues related to concrete Qt Components are presented in section IV.

II. Obstacles of Cross-platform Development with Qt Components

We have made an analysis of API of Qt Components 1.1 for Symbian and Harmattan platforms and defined four categories of potential problems when using them for cross-platform development, described in details in the following subsections of this section.

The issues for a concrete component may refer only to one of these classes, but it is common for components to have more than one potential problem belonging to different classes. Even APIs of such common elements like Button and CheckBox differ.

A. Components implemented only for one of the target platforms

The first class of issues emerges, because there are different component sets available for each platform. The missing components can be further subdivied into two categories.

1) Components belonging to subsystems, which were implemented solely for one platform: Examples of such components include various *Style* components: *LabelStyle*, *MenuStyle*, etc., which are available only for Harmattan and determine the look of specific components. Corresponding example from the Symbian platform is *InfoBanner* component, which provides a way to show a notification to the user in a non-obtrusive way.

2) Components potentially useful on either platform, but not implemented on one of them: For example, *ListHeading* component provides the ability to show a custom heading for the lists. It was implemented solely for Symbian platform. Another example is *ToolIcon* components, which enables a direct use of icons in a tool bar on Harmattan platform. Unfortunately, these components are not implemented on all platforms, and it seems there is no practical reason of such a decision.

B. Different APIs of the same components on different platforms

The second class of problems comes from the difference in APIs of the components having the same name. They may have different sets of properties and methods. The classification of these issues is quite similar to the one presented in the previous subsection.

1) Properties/methods related to subsystems implemented solely for one platform: It is the situation, when a subset of the distinct properties relates to the subsystems mentioned in the description of the case *A.1*. For example, several components, including *ToolBar*, *ButtonRow*, and others, provide *platformStyle* property on Harmattan platform.

2) Properties/methods potentially useful on either platform, but not implemented on one of them: For example, *Switch* component has *pressed* property on Symbian platform, but does not have it on Harmattan platform. This difference does not allow to trace the state when the user interacts with *Switch* element on Harmattan. The same problem applies to component methods. For example, *Slider* component has *valueChanged()* signal on Symbian platform but does not provide it on Harmattan platform. So it is needed to use property binding mechanism instead of signal handling to process the property value change.

3) Different inheritance relationship between the same components on different platforms: One more source of potential problems is that elements named the same way and providing similar API may be inherited from different components. It means that the set of inherited properties differ and the component may require specific environment to function correctly. A brightest example of such a problem is *Menu* component. On Symbian platform this component is implemented as a stand-alone component, functioning correctly in both screen orientations: portrait and landscape. On Harmattan platform this component is inherited from *Popup* element, therefore not providing satisfactory behavior in landscape orientation.

4) Components, methods, and properties having the same API and functionality, but different names: The example for this case is one specific property, which has different names in *SectionScroller* component on different platform. It is responsible for holding a *ListView* component to be scrolled. It has the name On Symbian platform it is *list* on Symbian and *listView* on Harmattan.

C. Issues of component implementation

Even if developer limits himself/herself to the usage of the components having the same interface, it cannot be guaranteed that the code will run without problems on both platforms. The reason is the existence of platform-specific bugs in implementation. One example of such a problem is the inability of *QueryDialog* component to display the last line of a multi-line message, which was present in Qt Components 1.1 release for Symbian [4].

D. Missing or differently named resources

This problem relates to the name of the resources, provided by the platforms. Both Symbian and Harmattan platforms have a notion of a theme, which describes the used colors, and contains specific resources, such as images used to display different parts of the interface. These resources are available to the external developers, who are encouraged to use them in applications. The problem of cross-platform development comes from the different naming scheme used for such resources on different platforms. For example, the name of tool bar image showing marker of the next page on Symbian platform is called *image://theme/toolbar-next* and *"image://theme/icon-m-toolbar-next-white"* on Harmattan platform.

E. QML limitations and poor naming convention

The names of Qt Components libraries for each platform are different, therefore to use these libraries it is needed to import *com.nokia.symbian* module on Symbian, and *com.nokia.meego* on Harmattan. Importing both modules in the same code cannot solve the problem, because QML language interpreter does not render QML elements with unresolved imports. Unfortunately, conditional imports in QML are also not supported.

III. QML Preprocessor

The classification of issues in the previous section indicates the need for a special tool or an approach allowing creation of cross-platform QML files. Such files should contain platform-specific information for both target platforms. The code implemented for one platform must not interfere with the normal execution flow on another platform. As it is shown in the last subsection of the previous section, the desired behavior is not supported by QML language. Though it might be possible to withdraw the declarative interface description and to create interface dynamically using JavaScript language, it would be highly inefficient.

We came up with the idea of preprocessor for QML files, which would allow to create platform-specific files based on the single QML file. The preprocessor handles the input QML file line-by-line. Each line of the file can be assigned to all platforms or to the specific one. By default, all lines are not platform-specific. If a line contains in-line comment in *@platform* format, it is associated with *platform*. There is no predefined list of platforms, but the lines of code targeted for the same platform must have identical labels. When the preprocessor starts, the text of the label is passed as an argument. For each input QML file the preprocessor generates the corresponding output QML file, containing only non-marked lines of the input file and the lines, corresponding to the selected platform. The generated files are further used to compile and deploy an application.

The Fig. 1 illustrates the operation of the preprocessor. The part (a) of the figure contains the description of *TabButton* component, which displays only one image. The component is designed to work on two target platforms. Lines of code marked with the *@symbian* label describe the creation of *Image* component, which displays the icon. This additional component is required on Symbian platform due to *TabButton* implementation, which does not provide *iconSource* property, though it is available on *Harmattan* platform. The part (b) of the figure demonstrates output of the preprocessor run on this file with *"symbian"* as a parameter. All platform-independent lines were left intact, but lines labeled with *@meego* were removed. The (c) part of the figure demonstrates the output of the preprocessor with *"meego"* given as a parameter.

The use of in-line comments allows not to interfere with Qt Quick platform at the runtime, but the presence of lines meant for different platforms prevents the usage of several QML tools. For example, such file cannot be opened in QMLViewer or modified in the visual QML modelling tool provided as a part of the Qt Creator IDE. The imposed restrictions are reduced by the fact that QML language is clear enough to edit QML files directly in a text editor, and processed files can be opened by the mentioned tools.

IV. Dealing with Obstacles

The QML preprocessor was successfully used for developing a couple of applications in our laboratory: Smart Conference clients [5] and Octotask [6], [7]. During the development process we encountered most of the issues described in Section II. In this section we demonstrate how those difficulties can be resolved with the use of QML Preprocessor described in Section III.

A. Components implemented only on one of the target platforms

There are two distinct approaches to deal with the absence of the components. According to the first one, developers have to implement required components themselves for the platform lacking for desired functionality. These elements should provide the same API as the missing components. Then developed files are bundled with the application only for the corresponding

```
TabButton {
    Image { //@symbian
        source: ”qrc:/icons/default” //@symbian
        ...
        fillMode: Image.PreserveAspectFit //@symbian
    } //@symbian
    iconSource: ”qrc:/icons/default” //@meego
}
```

(a)

```
TabButton {
    Image { //@symbian
        source: ”qrc:/icons/default” //@symbian
        ...
        fillMode: Image.PreserveAspectFit //@symbian
    } //@symbian
}
```

(b)

```
TabButton {
    iconSource: ”qrc:/icons/default” //@meego
}
```

(c)

Fig. 1. (a) Part of QML file containing preprocessor labels; (b) The same part of QML file preprocessed for Symbian platform; (c) Same part of QML file preprocessed for Harmattan platform.

platform. This can be achieved by adding the files to the appropriate resource list in the platform-specific section of qmake project file. Another way to deal with these problems is to implement required logic only for one platform and conceal it from others by the means of the preprocessor.

The first approach requires some extra investments in the first place, but it can be used if such functionality considered to be useful. It can also be easily reused in the future applications. For example, it should be useful to port *ListItem* component from Symbian platform, because it provides a visual decoration and a set of animations, therefore forming a specific application interface. On the other hand, porting notification subsystem looks unreasonable, because Harmattan users will probably be frustrated by the unfamiliar data presentation.

B. Different APIs of the same components on different platforms

There are few ways to deal with the situation: support some behavior solely for one platform or to provide similar look and feel on both platforms. Each of them can be implemented by the creation of a custom components or by using the QML preprocessor. When using the first approach, developer selects the API version, which he is intended to use in the project, and creates custom components for each platform. On one platform the custom element simply proxies all properties and methods to the existing component, while its counterpart for another platform proxies properties and methods of the base component, and provide

```
ButtonRow {
    ...
    style: TabButtonStyle { } //@meego
    ...
}
```

(a)

```
ToolButton {
    iconSource: "toolbar−menu" //@symbian
    flat: true //@meego
    Image { //@meego
        anchors.centerIn: menuButton //@meego
        sourceSize.height: menuButton.height //@meego
        source: ":images/menu.png" //@meego
    } //@meego
}
```

(b)

Fig. 2. Use of QML Preprocessor to support custom look of ButtonRow component on Harmattan platform. (a) Part of a QML file showing an emulation of iconSource property missing on Harmattan platform. (b)

additional methods to match the desired version of the API. This approach enforces reuse of the code, but makes the client code less readable, because element names differ from the base ones, and their role might be unclear. The developer needs to properly configure application build environment to include correct version of the element for corresponding platform.

With the use of preprocessor a developer can provide support solely for one platform by implementing the functionality for the platform and by placing proper labels on corresponding lines. We have used this approach in various places to customize the look of widgets on Harmattan platform with the use of *style* property. In case of having different property names or a lack of the property on one platform, the developer implements functionality separately for both platforms and places marks on corresponding lines of code. The resulting component may be created as a separate component, which provides unified API for both platforms.

The proposed approach was used to substitute *iconSource* property in ToolButton element on Harmattan platform. This property is used to specify a path to the image, which should be shown on *ToolButton* component. It is available on Symbian platform, so corresponding line is added and *@symbian* mark is placed on that line. The substitution for this property is the *Image* component, which is created on top of *ToolButton* element. The path to the image is setup as the value of the *source* property of embedded component. All lines, on which the nested *Image* component is declared, are marked with *@meego* label. Parts of QML files describing both cases are shown in Fig. 2.

C. Issues of component implementation

When the developer uses a component, which does not implement the declared functionality on one of the platforms due to an implementation error, it can be considered as the element having different API for distinct platforms. Therefore, the developer can use one of the proposed approaches for the previous class of problems. We faced the problem with the

implementation of *QueryDialog* component on Symbian platform [4], which did not display the last line of the message. The solution for this problem was to add an empty line to the dialog message only on Symbian.

D. Missing or differently named resources

In order to deal with this kind of problems, the developer needs to design a stable naming scheme for the required resources and used constants. The latter can be implemented as a special component, holding necessary information as the value of properties. This way, all components, which require resources or use some constant, will be bound with the properties of the proposed component. With the use of QML preprocessor such information can be placed into a single component, having two sets of properties for both platforms isolated from each other by the means of platform labels. Without the preprocessor the developer would need to manage two different components, which provide required constants through identical API for each platform. We have used the first approach and introduced *UiConstants* component.

E. QML limitations and poor naming convention

Qt Quick Components modules have different names for each target platform, which forces the developer to import corresponding module, when designing QML files for one platform, and to import distinct module for another platform. Developer cannot import both modules simultaneously, because one of the imports could not been resolved. This problem can be solved by the creation of a special empty modules having same names with the Qt Quick Components modules. When the build is done for Symbian platform the empty module named after the Harmattan module is added to the project and vice-versa. This way there would be no problem of missing module during the execution of such application. Another way to solve the problem is to use the QML preprocessor and place platform marks on import statements for the corresponding platforms. This would prevent appearance of the inappropriate imports in the production code.

V. Conclusion

Despite the fact that the task of cross-platform development involving Qt Quick Component looks rather difficult, our work showed that it is possible and beneficial, as it allows to share the same code base across the platforms.

As the use of the preprocessor extends the number of platforms the application running, it is required to appropriately test all the functionality of the application on all target platforms. In this case some kind of automatic testing framework is highly desirable.

There is Lighthouse technology, which allows external developers to port Qt framework to other platforms. One of the fastest developing project based on this technology is Necessitas [8], which is targeted to provide Qt framework and required infrastructure on Android OS. As a part of the initiative, Qt Quick Components library is ported there too [9]. The port is based on components developed for Symbian OS. Currently there are quite no changes of API except for module name, but they may appear in the future, so QML preprocessor based approach could be useful in this situation.

Nowadays, Necessitas has multiple architectural and practical limitations, which prevent its usage in production environment, but, as a proof of concept, we have successfully added support for Android platform into Smart Conference client project [5]. The source code of this

project, as well as source code of the project Octotask, which also uses QML preprocessor, is available from corresponding repositories (see project pages [5], [7]).

Our experience in development of the mentioned projects shows that no more than 10 percent of source code has to be implemented separately across the platforms. Such a small value shows the effectiveness of the used approach.

In conclusion, we would like to mention that developers of Qt Quick Components are trying to reduce segmentation of the library, which means that the number of problems presented in this section should decrease in future releases [10].

REFERENCES

[1] "Qt Quick Colibri project," Feb. 2012. [Online]. Available: http://projects.developer.nokia.com/colibri.
[2] "Qt Quick Components project," Feb. 2012. [Online]. Available: http://qt.gitorious.org/qt-components.
[3] D. Thomas and A. Hunt "The Pragmatic Programmer: From Journeyman to Master," Addison-Wesley Professional, 1999.
[4] "QueryDialog never shows last line of message," Feb. 2012. [Online]. Available: https://bugreports.qt-project.org/browse/QTCOMPONENTS-1090.
[5] "Smart Conference clients project," Feb. 2012. [Online]. Available: http://yar.fruct.org/projects/sc-clients.
[6] A. Vasilev and I. Paramonov. "Concept of Octotask — Multi-source Task Collector and Manager," in *Proceedings of the 9th Conference of Open Innovations Community FRUCT and 1st Regional MeeGo Summit Russia-Finland*, 2011, p. 244.
[7] "Octotask project," Feb. 2012. [Online]. Available: http://yar.fruct.org/projects/octotask.
[8] "Necessitas project," Feb. 2012. [Online]. Available: http://sourceforge.net/p/necessitas/.
[9] "Android Qt Components," Feb. 2012. [Online]. Available: http://qt.gitorious.org/~koying/qt-components/android-qt-components/.
[10] "Define a baseline Item set as common API between different component style implementations," Feb. 2012. [Online]. https://bugreports.qt-project.org/browse/QTCOMPONENTS-200.