Capítulo

9

Desenvolvimento Rápido de Aplicações Móveis Utilizando a Linguagem Declarativa QML

Ricardo Erikson V. S. Rosa, Adriano M. Gil, Paulo R. B. Mendonça, Cícero F. F. Costa Filho e Vicente F. Lucena Jr.

Abstract

The emerging software development technologies have their focus on reducing the time-to-market of new products. In the mobile applications context, reduce time-to-market is critical to maximize ROI. QML (Qt Meta-object Language) is part of a declarative UI framework known as Qt Quick which aims to facilite the rapid development of applications. This course aims to present the main components and features of the QML language and Qt Quick UI framework addressed to the development of mobile applications.

Resumo

As novas tecnologias de desenvolvimento de software buscam meios de reduzir o tempo excessivo gasto na criação de novos produtos. No contexto de aplicações móveis, reduzir o tempo que um produto leva para chegar ao mercado é crítico para maximizar o retorno sobre investimento. QML é uma linguagem declarativa que facilita o desenvolvimento rápido de interfaces do usuário (UIs) e consequentemente reduz o tempo de desenvolvimento. Este minicurso tem por objetivo apresentar os principais componentes e características da linguagem QML e do framework Qt Quick com foco em aplicações móveis.

9.1. Introdução

Desenvolvedores de aplicações móveis buscam meios de reduzir o tempo excessivo gasto na criação e publicação de novos produtos. No mercado das *app stores*, as lojas online de aplicativos, o desenvolvedor precisa considerar três parâmetros de tempo: (1) tempo de desenvolvimento, (2) *time to shelf* (intervalo de tempo entre a submissão de uma aplicação e sua publicação para compra na loja) e (3) *time to payment* (período de tempo entre a venda da aplicação e os rendimentos chegarem até o desenvolvedor). A redução

do intervalo de tempo compreendido entre o início do desenvolvimento e o retorno dos rendimentos é um fator crítico para maximizar o ROI.

Levando em consideração o tempo de desenvolvimento, a rapidez na codificação e na prototipagem dos aplicativos é apontada como uma das principais razões técnicas para que os desenvolvedores escolham uma plataforma de desenvolvimento [10]. Os resultados de codificação e prototipagem são muitas vezes apresentados na forma de interfaces do usuário (UIs). No entanto, a dificuldade de criação dessas UIs, onde os usuários têm que aprender novas APIs inteiramente sem levar em conta as experiências de desenvolvimento em outras plataformas, é destacada como um dos problemas comumente enfrentado pelos desenvolvedores em várias plataformas [9, 10].

QML (*Qt Meta-objects Language*) é uma linguagem declarativa utilizada no kit de desenvolvimento Qt Quick (***Qt User Interface Creation Kit***) [8] que faz parte do framework Qt[1]. QML é utilizada no desenvolvimento de aplicativos *cross-platform* e busca facilitar o projeto e a implementação de UIs para dispositivos móveis através da rapidez na codificação e na prototipagem. O estilo de programação da linguagem QML é baseado nas linguagens CSS (*Cascading Style Sheets*) e JavaScript, tornando-se de aprendizado rápido e fácil para programadores C, Qt/C++, Java e principalmente desenvolvedores web.

Diferentemente de outras linguagens de programação que utilizam a abordagem imperativa para construir as UIs, QML utiliza o paradigma declarativo. A utilização desse paradigma permite que os desenvolvedores descrevam UIs declarando somente os objetos que devem ser exibidos, ou seja, sem descrever o fluxo de controle ou sequência de ações que devem ser executadas para criar a interface. Enquanto a UI é criada de maneira declarativa, a lógica da aplicação pode ser escrita de maneira imperativa através das linguagens JavaScript e C++. Isso facilita a separação entre a UI e a lógica da aplicação, permitindo que projetos Qt/C++ baseados no padrão *model-view-controller* (MVC) sejam facilmente portados para QML [1, 2].

Uma das principais características da linguagem QML é a flexibilidade na criação das UIs. Essa flexibilidade é possível graças à possibilidade de adicionar e manipular elementos como formas, imagens e texto. Outros elementos como estados, transições, animações, transformações, efeitos gráficos e elementos de interação permitem a criação de aplicações mais sofisticadas em QML, principalmente quando utilizados em conjunto com JavaScript e C++.

Este capítulo tem o objetivo de apresentar a linguagem QML e sua utilização no desenvolvimento rápido de aplicações móveis. Os principais características da linguagem são apresentados na Seção 9.2. A utilização dos elementos no tratamento de eventos é demonstrada na Seção 9.3. A Seção 9.4 mostra os principais elementos utilizados na criação de animações. Os mecanismos de criação de componentes customizados são descritos na Seção 9.5. As particularidades das principais plataformas de desenvolvimento são mostradas na Seção 9.6 e, por fim, as considerações finais são feitas na Seção 9.7.

[1]Qt é um framework de desenvolvimento de aplicações *cross-platform* que foi desenvolvido em C++ e está disponível para Windows, Linux e Mac OS X. O framework disponibiliza um amplo conjunto de APIs que podem ser utilizadas no desenvolvimento para plataformas móveis embarcadas e desktop.

9.2. Linguagem Declarativa QML

Muitas plataformas móveis disponibilizam APIs, em linguagens de programação comumente conhecidas (por exemplo Symbian C++, Qt/C++, Java, e Objective-C), para que os desenvolvedores criem as UIs dos aplicativos. Porém, muitas vezes o custo de aprendizado de uma API inteira, considerando também as particularidades da sintaxe de cada linguagem, é bastante alto [9]. Além disso, o paradigma imperativo, que é frequentemente empregado nas linguagens de programação modernas, faz com que o processo de criação das UIs se torne árduo e demorado [12]. Baseando-se nesses fatos, o desenvolvimento *cross-platform* utilizando plataformas que possibilitem baixo custo de aprendizado e rápida prototipagem parece ser uma alternativa desejável no desenvolvimento de aplicativos móveis.

A linguagem declarativa QML fornece um ambiente de programação bastante simples e flexível, porém robusto. Ela dispõe de um amplo conjunto de componentes, para a criação das UIs, e de mecanismos de integração (*bindings*) com as linguagens JavaScript e Qt/C++. Os objetos da UI dos aplicativos QML são especificados por elementos que possuem tipo e propriedades (no estilo nome:valor, como em CSS), o que torna a linguagem intuitiva e de fácil aprendizado tanto para designers de UI quanto para programadores. Para ilustrar isso, as Listagens 9.1 e 9.2 apresentam uma comparação entre as linguagens Qt/C++ e QML para a criação de um simples elemento visual.

O código Qt/C++ necessário para desenhar um retângulo é apresentado na Listagem 9.1. Esse código exibe o retângulo na UI e define propriedades como largura, altura e cor.

```
1 void paint(QPainter *painter,
2   const QStyleOptionGraphicsItem *option,
3   QWidget *widget){
4     QRect rect(0, 0, 200, 100);
5     painter->setBrush(QBrush(Qt::green));
6     painter->drawRect(rect);
7 }
```

Listagem 9.1. Código Qt/C++ necessário para exibir um retângulo.

A Listagem 9.2 exibe o código QML necessário para desenhar o mesmo retângulo. O retângulo criado consiste em um único objeto do tipo Rectangle com 3 propriedades (width, height e color).

```
1 import QtQuick 1.0
2 Rectangle {
3     width: 200
4     height: 100
5     color: "green"
6 }
```

Listagem 9.2. Código QML necessário para exibir um retângulo.

9.2.1. Ambiente de Desenvolvimento

O ambiente necessário para o desenvolvimento de aplicativos em QML é disponibilizado juntamente com o QtSDK [5], que pode ser obtido em http://qt.nokia.com/downloads. O QtSDK inclui o Qt Quick e disponibiliza a ferramenta Qt QML Viewer [6], que é muito utilizada no desenvolvimento e depuração de código QML, mas não deve ser utilizada em ambiente de produção. Outra opção disponível no SDK é a utilização da IDE Qt Creator, que fornece um ambiente completo para o desenvolvimento de aplicativos em Qt/C++ e QML.

Uma aplicação QML é executada através da máquina de execução QML, também chamada de *QML runtime*, para ser executada. Existem duas maneiras de se iniciar essa máquina de execução: (1) a partir de uma aplicação Qt/C++ (utilizando a classe QDeclarativeView) ou (2) através da ferramenta Qt QML Viwer[2]. Aplicativos QML destinados ao usuário final (para instalação e utilização no dispositivo móvel) devem utilizar a primeira opção. Já os aplicativos quando ainda em processo de desenvolvimento podem ser testados utilizando o Qt QML Viewer.

Os exemplos apresentados nas próximas seções podem ser testados utilizando o Qt QML Viewer. Porém, mais adiante neste capítulo, será apresentado como invocar aplicativos QML utilizando código Qt/C++.

9.2.2. Propriedades e Tipos de Dados

Os objetos QML são especificados por meio de seus elementos e cada elemento possui um conjunto de propriedades. Essas propriedades são formadas por pares nome-valor (por exemplo, color:“blue”) e assumem uma variedade de tipos de dados que podem ser referências para outros objetos, strings, números, etc. Em QML, as propriedades são fortemente tipadas, ou seja, se uma propriedade possui um tipo específico então um valor de tipo diferente não pode ser atribuído à ela. A Tabela 9.1 apresenta alguns dos tipos de dados comumente utilizados em propriedades QML.

Objetos definidos em QML podem ser referenciados por outros objetos por meio da propriedade id. Essa propriedade é definida explicitamente pelo programador para identificar um objeto dentro do escopo QML. Dessa maneira, o valor de uma propriedade de um determinado objeto podem ser externamente acessado por <id>.<propriedade>, como na linha 5 da Listagem 9.3 com obj2.width.

Uma propriedade pode ser expressada em função de outra propriedade do mesmo objeto (ou de um objeto diferente, por meio de seu id) desde que possuam tipos compatíveis. Dessa maneira, se o valor da propriedade referenciada for alterado durante a execução aplicação, a máquina de execução QML recalcula a expressão e atualiza o valor da propriedade que fez a referência. Por exemplo, na Listagem 9.3 a propriedade height é definida pela expressão obj2.width*2. Então, se o valor da propriedade width for alterado, o valor de height será automaticamente atualizado, alterando também a aparência visual do objeto.

QML permite que novas propriedades sejam declaradas na definição de um objeto. Ainda na Listagem 9.3 (linha 7), a propriedade area, que não faz parte da definição

[2]Para testar o aplicativo é necessário executar o seguinte comando: $ qmlviewer arquivo.qml

Tabela 9.1. Alguns tipos utilizados no sistema de tipagem da linguagem QML.

Tipo	Descrição
color	Nomes de cores especificadas no padrão SVG [11] entre aspas. Os valores também podem ser especificados nos formatos "#RRGGBB" ou "#AARRGGBB".
bool	Pode assumir os valores true ou false.
date	Data especificada no formato "YYYY-MM-DD".
time	Horário especificado no formato "hh:mm:ss".
font	Encapsula as propriedades de uma instância do tipo QFont do Qt.
int	Representa valores inteiros.
double	Possui ponto decimal e seus valores são armazenados com precisão dupla.
real	Representa um número real.
list	Representa uma lista de objetos.
point	Representa um ponto através das coordenadas x e y.
rect	Consiste na representação dos atributos x, y, width e height.
size	Consiste na representação dos atributos width e height.
string	Texto livre entre aspas.
url	Representa a localização de algum recurso, como um arquivo, através de seu endereço.

original do tipo Rectangle, é definida como sendo o produto width*height. O valor dessa propriedade é atualizado pela máquina de execução QML sempre que os valores de width e height são alterados.

```
1 import QtQuick 1.0
2 Rectangle {
3     id: obj1
4     width: 200
5     height: obj2.width*2
6     color: "#008000"
7     property real area: width*height
8 }
```

Listagem 9.3. Exemplos da utilização de propriedades na linguagem QML.

9.2.3. Elementos Básicos

A linguagem QML oferece um conjunto de elementos que permitem a rápida e fácil inclusão de objetos nos aplicativos. Esses elementos podem ser aninhados permitindo a criação de um relacionamento do tipo pai-filho (*parent-child*) entre eles. Esse tipo de relacionamento pode ser muito útil para a criação de layouts baseados em pontos de ancoragem, que será apresentado na Seção 9.2.4.

Uma lista dos elementos mais básico comumente encontrados em QML está disponível na Tabela 9.2. Esses elementos, através de suas propriedades e seus sinais [3],

[3]Sinais (de sinais e *slots*) é uma característica muito importante no tratamento de eventos do framework

Tabela 9.2. Alguns elementos básicos disponíveis na linguagem QML.

Elemento	Descrição
Item	O elemento mais básico da linguagem QML. Embora não possua aparência visual, ele é utilizado como base de todos os elementos visuais.
Rectangle	É um dos elementos mais básicos para a criação de aplicações em QML. É utilizado para preenchimento de áreas e é frequentemente utilizado como base para o posicionamento de outros elementos.
Image	O elemento é utilizado para exibir imagens na UI. Todos os formatos suportados pelo Qt podem ser utilizados no elemento Image, incluindo PNG, JPEG e SVG.
Text	É utilizado para adicionar texto na UI.
TextInput	Esse elemento possibilita a entrada de texto no aplicativo, podendo restringir o formato dos dados de entrada através da utilização de uma máscara de validação.
MouseArea	É um elemento invisível, frequentemente utilizado em conjunto com elementos visíveis, no tratamento de eventos de entrada com o ponteiro do mouse ou toques na tela.

possibilitam a criação das UIs e o tratamento de eventos em aplicativos móveis. Uma lista mais completa dos elementos QML pode ser encontrada em [4].

9.2.4. Layouts Baseados em Pontos de Ancoragem

O layout baseado em pontos de ancoragem é um poderoso recurso proporcionado pela linguagem QML e é muito útil no posicionamento dos objetos na UI dos aplicativos. Cada elemento QML possui basicamente seis pontos invisíveis de ancoramento: left, right, top, bottom, horizontalCenter e verticalCenter. Esses pontos possibilitam a criação de relacionamentos entre as linhas de ancoragem de diferentes objetos. Em outras palavras, esses pontos facilitam o posicionamento de um objeto em relação a outros objetos adjacentes.

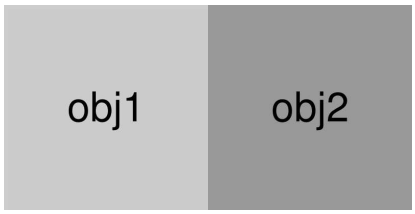
A Figura 9.1 apresenta um exemplo da utilização de pontos de ancoragem com dois elementos do tipo Rectangle. O código apresentado a esquerda (Figura 9.1(a)) realiza a ancoragem da borda esquerda do objeto obj2 à borda direita do objeto obj1. Dessa maneira, obj2 sempre fica localizado à direita de obj1. O resultado visual da utilização desses pontos de ancoragem é apresentado na Figura 9.1(b).

Múltiplos pontos de ancoragem podem ser especificados por objeto. Esse tipo de abordagem permite maior flexibilidade no posicionamento dos objetos assim como melhor ajuste da UI do aplicativo. Isso pode ser muito útil na adaptação de aplicativos para execução em dispositivos com tamanhos de tela diferentes. Uma das abordagens utilizadas por designers gráficos é criar áreas de flutuação (que variam de tamanho) para facilitar o reajuste dos objetos da UI em tamanhos de tela variados. A utilização de ancoragem permite que os valores de largura e/ou altura dessas áreas sejam ajustados sem que sejam

Qt. Os sinais são emitidos com a intenção de comunicar eventos que ocorrem na aplicação. Os *slots*, por sua vez, são funções que são chamadas respondendo aos sinais, ou seja, aos eventos.

```
1 Rectangle {
2     id:obj1; ...
3     Text {text: "obj1"; ...}
4 }
5 Rectangle {
6     id:obj2; ...
7     Text {text:  "obj2"; ... }
8     anchors.left: obj1.right
9 }
```

(a) Código



(b) Resultado

Figura 9.1. Ancoragem da borda direita do objeto obj1 à borda esquerda do objeto obj2.

definidos explicitamente.

A Figura 9.2(a) apresenta o código para a criação de uma área que é redimensionada dependendo do posicionamento dos outros objetos. Na definição do objeto obj2, o ponto top foi ancorado ao bottom do objeto obj1, e o ponto bottom foi ancorado ao top do objeto obj3. Como resultado, a propriedade height do objeto obj2 é dependente do posicionamento dos objetos obj1 e obj3, não necessitando defini-la explicitamente. O resultado visual dessa ancoragem é apresentado na Figura 9.2(b).

```
 1 Rectangle {
 2     id: obj1; ...
 3     x: 0; y: 0; width: 120
 4 }
 5 Rectangle {
 6     id: obj2; ...
 7     anchors.top: obj1.bottom
 8     anchors.bottom: obj3.top
 9 }
10 Rectangle {
11     id: obj3; ...
12     x: 0; y: 200; width: 120
13 }
```

(a) Código

(b) Resultado

Figura 9.2. Exemplo da utilização de ancoragem para a criação um objeto que varia a propriedade height automaticamente.

Uma propriedade de ancoragem muito útil e frequentemente utilizada para o preenchimento de objetos é o anchors.fill. Essa propriedade convenientemente fixa as âncoras left, right, top e bottom de um objeto às âncoras left, right, top e bottom de um objeto alvo. Essa propriedade é muitas vezes utilizada pelo tipo MouseArea para criar uma área clicável com as mesmas dimensões de um outro objeto.

Outra propriedade de conveniência bastante utilizada é o anchors.centerIn. Essa propriedade é utilizada para automaticamente centralizar um objeto em relação à outro objeto. A propriedade anchors.centerIn fixa as âncoras verticalCenter e horizontal-

Center de um objeto às âncoras verticalCenter e horizontalCenter de outro objeto.

9.2.5. Criando a Primeira Aplicação QML

Como explicado anteriormente na Seção 9.2.1, para que seja instalada e executada no dispositivo móvel, uma aplicação QML precisa ser executada a partir de uma aplicação Qt/C++ utilizando a classe QDeclarativeView. A IDE Qt Creator possui *templates* que facilitam a criação de projetos para executar aplicações QML. No lado esquerdo da tela de criação de novos projetos do Qt Creator, deve ser escolhido um projeto do tipo *Qt Quick Project*, e no lado direito, um *template* do tipo *Qt Quick Application*.

Após a escolha do nome do projeto e sua localização, é necessário escolher o *target*. Dentre as opções de *target* apresentadas, normalmente são escolhidas duas opções para auxiliar no desenvolvimento de aplicações móveis: *Desktop* e *Remote Compiler*. A opção *Desktop* é utilizada para compilação e execução da aplicação no próprio desktop durante o desenvolvimento. Já a opção *Remote Compiler* é utilizada para a compilação dos aplicativos para plataformas móveis específicas.

No projeto criado, o arquivo principal da aplicação (*main.cpp*) possui o código que executa o arquivo QML principal, conforme apresentado na Listagem 9.4. Nesse exemplo, o objeto viewer do tipo QmlApplicationViewer é configurado e define o arquivo QML principal como sendo "qml/ERCEMAPI/main.qml" (linha 10). Em seguida, o método showExpanded() do objeto *viewer* apresenta a o arquivo QML na tela do aplicativo, conforme apresentado na linha 11.

O tipo QmlApplicationViewer é um tipo definido no *template* de criação do projeto e é do tipo QDeclarativeView. O QmlApplicationViewer é uma classe de conveniência criada com configurações específicas de código para cada plataforma móvel, como orientação de tela e *debug*.

```
 1 // Arquivo: main.cpp
 2 #include <QtGui/QApplication>
 3 #include "qmlapplicationviewer.h"
 4
 5 int main(int argc, char *argv[])
 6 {
 7     QApplication app(argc, argv);
 8     QmlApplicationViewer viewer;
 9     viewer.setOrientation(
10         QmlApplicationViewer::ScreenOrientationAuto);
11     viewer.setMainQmlFile(QLatin1String("qml/ERCEMAPI/main.qml"));
12     viewer.showExpanded();
13     return app.exec();
14 }
```

Listagem 9.4. Arquivo Qt/C++ principal de uma aplicação QML executável no dispositivo móvel.

Na Listagem 9.5 encontra-se o arquivo "main.qml", que possui código QML principal da aplicação. Esse código possui uma aplicação simples que exibe uma mensagem "Olá Mundo QML" e exibe uma imagem com o logo do Framework Qt ao se clicar em

botão. Logo na primeira linha do arquivo, é declarada a instrução import QtQuick 1.0, utilizada para importar os principais elementos da linguagem QML. A seguir é declarado um elemento do tipo Rectangle como a base do aplicativo. É interessante observar que o elemento Item também pode ser utilizado para criar essa base para a construção do aplicativo. Nas linhas seguintes, o restante da aplicação é construída declarando-se objetos dos tipos Image, Rectangle, Text e MouseArea.

```
 1 // Arquivo: main.qml
 2 import QtQuick 1.0
 3
 4 Rectangle {
 5     width: 640; height: 360
 6     color: "white"
 7     Image {
 8         id: img
 9         y: 250
10         source: "qt.png"
11         visible: false
12         anchors.horizontalCenter: parent.horizontalCenter
13     }
14     Rectangle {
15         color: "limegreen"; width: 150; height: 50; radius: 5
16         anchors.horizontalCenter: parent.horizontalCenter; y: 50
17         Text {
18             anchors.centerIn: parent;
19             font { family: "Helvetica"; pointSize: 20 }
20             text: "Clique aqui"
21         }
22         MouseArea {
23             anchors.fill: parent
24             onClicked: {
25                 msg.text = "Olá Mundo QML"
26                 img.visible = true
27             }
28         }
29     }
30     Text {
31         id: msg;
32         font { family: "Helvetica"; pointSize: 40 }
33         anchors.centerIn: parent; text: ""
34     }
35 }
```

Listagem 9.5. Arquivo QML principal com o código da aplicação.

9.3. Tratamento de Eventos

Um dos pontos mais importantes no desenvolvimento de software para dispositivos móveis é o tratamento de eventos, pois é através desses eventos que ocorre a interação entre o usuário e a aplicação. Nesta seção serão abordadas as principais maneiras de interação com o usuário através do mecanismo de tratamento de eventos utilizado na linguagem QML e no framework Qt Quick.

9.3.1. Sistema de Eventos Baseado em Sinais

O tratamento de eventos da linguagem QML é similar ao mecanismo de sinais e *slots* disponível no framework Qt. Em QML os sinais são emitidos para notificar a ocorrência de eventos e estão conectados a *slots* específicos (também conhecidos como manipuladores de sinais ou *signal handlers*). Esses *slots*, por sua vez, permitem que código JavaScript seja executado em resposta à um evento. Dessa maneira, sempre que um determinado sinal associado a um evento for emitido, o *slot* associado a esse sinal é chamado e o código JavaScript contido nele é executado.

Cada elemento QML possui seus próprios *slots*, cada um associado a um sinal ou mesmo à uma propriedade. No caso dos sinais, os *slots* seguem uma sintaxe padrão do tipo: on<Sinal>. Por exemplo, o tipo MouseArea possui o sinal clicked e um slot associado chamado onClicked. De maneira semelhante, cada propriedade possui um *slot* associado às mudanças de valor através da seguinte sintaxe: on<Propriedade>Changed. Por exemplo, o *slot* onWidthChanged refere-se ao *slot* da propriedade width. Assim, sempre que ocorrerem alterações no valor dessa propriedade, seu slot correspondente será chamado.

O código exibido na Listagem 9.6 apresenta um exemplo de tratamento de evento de clique em um elemento do tipo MouseArea. Esse código fecha a aplicação executando o comando JavaScript quit() disponível na variável global Qt.

```
1 MouseArea {
2     anchors.fill: parent
3     onClicked: {
4         Qt.quit();
5     }
6 }
```

Listagem 9.6. Tratamento de evento de clique no elemento MouseArea.

Ao declarar novos sinais ou propriedades, o framework Qt Quick automaticamente cria *slots* seguindo as convenções de nomenclatura apresentadas anteriormente. Logo, é possível construir um sistema próprio de eventos, estendendo os sinais e *slots* básicos providos pelo framework Qt Quick. Por exemplo, ao implementar um teclado virtual, seria interessante que a assinatura do sinal tivesse um parâmetro para enviar o valor da tecla pressionada ao slot quando o sinal for emitido.

O código exibido na Listagem 9.7 mostra como um teclado virtual poderia ser implementado em QML. O elemento Rectangle não possui um sinal que é emitido na ocorrência de eventos de clique, e tão pouco um *slot* para tratar esses eventos. Dessa maneira, um sinal clicked(string value) é declarado recebendo como argumento um valor do tipo string (linha 5). Ao declarar esse sinal, o *slot* onClicked é criado automaticamente no objeto button do tipo Rectangle. No *slot* onClicked do elemento MouseArea, o sinal button.clicked(string value) é emitido e passa como argumento o valor buttonText.text (linha 13). No *slot* onClicked do objeto button, essa propriedade pode ser acessada através do parâmetro value passado pelo sinal clicked(string value), conforme apresentado na linha 15.

```
 1 import QtQuick 1.0
 2
 3 Rectangle {
 4     id: button; ...
 5     signal clicked(string value)
 6     Text {
 7         id: buttonText
 8         anchors.centerIn: parent
 9         text: "A"
10     }
11     MouseArea {
12         anchors.fill: parent
13         onClicked: button.clicked(buttonText.text)
14     }
15     onClicked: console.log(value)
16 }
```

Listagem 9.7. Trecho de código da implementação de um teclado virtual com sinais e *slots* customizados.

9.3.2. Elementos de Interação

O mais primário tipo de interação que um usuário espera ter com um software moderno se dá através de interações de mouse ou outro mecanismo apontador. Esse tipo de interação não limita-se somente aos cliques. Na verdade, existem vários outros eventos como botão pressionado, botão solto, movimento, etc. Com a crescente popularização das telas de toque em dispositivos móveis, como tablets e smartphones, outros tipos de interações (como gestos de *flick*, *swipe*, *pinch* e *drag*) vêm sendo absorvidas pelos usuários.

Na linguagem QML alguns componentes já implementam algum tipo de interação, como é o caso dos elementos a seguir: ListView, GridView e PathView, que permitem fazer o *scroll* por seus elementos; TextInput e TextEdit, que possibilitam a inserção e edição de texto; Flickable e PinchArea, que lidam com a interação através de gestos de *flick* e *pinch*; Flipable e MouseArea, que tratam dos eventos de mouse; e finalmente Keys e FocusScope, referentes aos eventos de teclado. Nas seções a seguir, serão apresentados os tipos de interação mais básicos que podem ser implementados com a linguagem QML: eventos de clique (MouseArea) e entrada de texto (TextInput).

9.3.2.1. MouseArea

O MouseArea trata-se de um elemento para o tratamento de eventos de clique. Ele possui todos os sinais e *slots* necessários para a construção de qualquer funcionalidade baseada em eventos de mouse. O MouseArea é um elemento invisível e sua utilização é, tipicamente, associada a um elemento visível.

O elemento MouseArea permite capturar uma variedade de eventos através de seus *slots*. Os *slots* apresentados na Tabela 9.3 são os mais comuns em dispositivos móveis e podem ser amplamente utilizados no tratamento de eventos em aparelhos que possuem tela de toque. O exemplo exibido na Listagem 9.8 apresenta o código para o tratamento de eventos de clique (onClicked) e pressionamento e liberação (onPressAndHold) em

aplicativos que possuam tela de toque. O *slot* chamado para cada sinal escreve o tipo de evento no console do aplicativo. Os outros eventos apresentados na Tabela 9.3 são tratados de maneira semelhante.

Slot	**Descrição**
onCanceled	Chamado quando um evento de mouse é cancelado, seja porque o evento não foi aceito ou porque foi interceptado por outro elemento.
onClicked	Chamado quando ocorre um clique, ou seja, um pressionamento seguido de liberação.
onDoubleClicked	Chamado quando ocorre um duplo clique, ou seja, um pressionamento seguido de uma liberação seguida de outro pressionamento.
onPressAndHold	Esse *slot* é chamado quando existe um longo pressionamento (durante 800ms).
onPressed	O *slot* é chamado quando ocorre um pressionamento.
onReleased	Chamado quando ocorre uma liberação do ponteiro.

Tabela 9.3. Principais *slots* associados aos eventos do elemento MouseArea.

```
 1 import QtQuick 1.0
 2
 3 Item {
 4     width: 360
 5     height: 640
 6
 7     MouseArea {
 8         anchors.fill: parent
 9         onClicked: console.log("Evento: click")
10         onPressAndHold: {
11             console.log("Evento: Press and hold")
12         }
13     }
14 }
```

Listagem 9.8. Tratamento de eventos de mouse com os *slots* onClicked e onPressAndHold.

Outra funcionalidade muito interessante proporcionada pelo MouseArea é o *drag*. Através do *drag*, o usuário pode arrastar elementos na tela. Esse evento é muito utilizado em *dock widgets*[4] e jogos. O exemplo exibido na Listagem 9.9 ilustra como o evento de *drag* pode ser alcançado facilmente usando o elemento MouseArea.

É importante observar que o layout baseado em pontos de ancoragem pode comprometer o tratamento de eventos de *drag*. Dependendo do tipo de ancoragem utilizado, o objeto pode ficar totalmente ancorado, impedindo sua movimentação na tela do aplicativo.

[4] *Dock widgets* é um painel de ícones que inicialmente aparece escondida nas bordas da tela de um aplicativo. Ao clicar e arrastar a *dock widgets*, o painel com os ícones é exibido, permitindo assim que esses ícones sejam clicados.

```
 1 import QtQuick 1.0
 2
 3 Rectangle {
 4     id: background
 5     color: "lightblue"
 6     width: 360
 7     height: 640
 8     ...
 9     Rectangle {
10         id: dragable
11         color: "white"
12         width: 50
13         height: 50
14         ...
15         MouseArea {
16             anchors.fill: parent
17             drag.target: dragable
18         }
19     }
20 }
```

Listagem 9.9. Exemplo de implementação de um evento de *drag*.

9.3.2.2. Entrada de Dados

Na linguagem QML, os elementos disponíveis para realizar entrada de dados são TextInput e TextEdit. Ambos são utilizados para entrada de texto, porém o TextInput mostra uma única linha de texto editável não formatado, enquanto o TextEdit exibe várias linhas de texto editável e formatado.

O texto digitado no elemento TextInput pode ser restringido através da utilização da propriedade validator ou inputMask. Utilizando as restrições de entrada, o elemento TextInput somente aceitará o texto que esteja em um estado aceitável ou, no mínimo, intermediário. Alguns validadores suportados são IntValidator, DoubleValidator e RegExpValidator.

A Listagem 9.10 apresenta um trecho de código demonstrando a utilização da propriedade validator para a validação de texto. Nesse exemplo, a propriedade utiliza o tipo DoubleValidator com as seguintes propriedades: bottom (menor valor assumido pelo campo), top (maior valor assumido pelo campo) e decimals (número máximo de dígitos após o ponto decimal). O *slot* onTextChanged verifica o valor da propriedade acceptableInput e modifica a cor de fundo do campo de entrada para verde claro quando o texto for válido ou para vermelho claro, caso contrário.

Outra possibilidade interessante do elemento TextInput é a utilização da propriedade echoMode, que especifica como o texto deve ser apresentado. Essa propriedade pode assumir as seguintes opções: TextInput.Normal (exibição normal), TextInput.Password (exibe asteriscos e é normalmente utilizado para exibição de senhas), TextInput.NoEcho (não exibe o texto) e TextInput.PasswordEchoOnEdit (com exceção do caractere atual, todos os outros são exibidos como asterisco).

```
 1 import QtQuick 1.0
 2
 3 Rectangle {
 4     id: background
 5     width: 200
 6     height: 70
 7     TextInput {
 8         focus: true; horizontalAlignment: TextInput.AlignHCenter
 9         font {family: "Helvetica"; pointSize: 20}
10         anchors { fill: parent; topMargin: 20 }
11         validator: DoubleValidator {
12             bottom: 0.0; top: 99.99; decimals: 2
13         }
14         onTextChanged: {
15             if (acceptableInput) {
16                 background.color = "lightgreen"
17             } else {
18                 background.color = "indianred"
19             }
20         }
21     }
22 }
```

Listagem 9.10. Exemplo do uso da propriedade validator no elemento TextInput.

9.4. Criando Animações com Estados e Transições

As propriedades de objetos QML podem ser manipuladas através de código JavaScript, permitindo assim que animações sejam criadas de maneira imperativa. A criação de animações através de código JavaScript é muitas vezes uma tarefa complexa para os programadores. QML disponibiliza um conjunto de elementos baseados em estados e transições, que quando associados a outros elementos, facilitam a criação de animações nas aplicações. As seções a seguir apresentam os conceitos e elementos relacionados a estados, transições e animações em QML.

9.4.1. Estados em QML

Em QML, estados são coleções de configurações definidas através do elemento State. Um objeto alcança um estado quando as mudanças realizadas nas suas propriedades representam um dos estados presentes em sua coleção de estados. A propriedade states armazena essa coleção (representado através de uma lista de objetos do tipo State) e representa todos os estados possíveis de um objeto.

A simples declaração de um elemento automaticamente cria um estado padrão, chamado de estado base, definido como “” (*string* vazia). Esse é o estado em que um objeto se encontra quando não representa nenhum dos estados definidos em sua propriedade states. Outros estados podem ser declarados e a transição entre esses estados pode ser realizada facilmente modificando o valor da propriedade state (no singular), que armazena o estado atual de um objeto. O valor dessa propriedade pode ser modificado diretamente em um *slot* ou função escrita em JavaScript, como apresentado na Listagem 9.11.

O elemento State possui as seguintes propriedades: changes (lista de mudanças

```
1 onClicked:  {
2     if(obj.state == "clicado") {
3         obj.state = "";
4     }
5     else {
6         obj.state = "clicado";
7     }
8 }
```

Listagem 9.11. Exemplo da alteração do estado de um objeto utilizando código JavaScript diretamente.

a serem realizadas neste estado), extend (nome do estado pai), name (nome do estado) e when (condição para ativação do estado). A Listagem 9.12 apresenta um exemplo da utilização da propriedade when para a ativação do estado "clicado" do objeto rect. Nesse caso, a propriedade diz que esse estado só será ativado quando o objeto representado pelo MouseArea for clicado, ou seja, quando o sinal clicked do objeto mouse for emitido.

Na linha 12 da Listagem 9.12 também é possível observar a utilização do elemento PropertyChanges. Esse elemento é utilizado para modificar as propriedades de um objeto específico identificado em target durante o tempo em que um estado estiver ativado. Então, nesse exemplo, a propriedade color do objeto rect (target) é alterada para "red" enquanto esse objeto estiver no estado "clicado".

```
 1 import QtQuick 1.0
 2
 3 Rectangle {
 4     id: rect; color: "green";  width: 120; height: 120
 5     MouseArea { id: mouse; anchors.fill: parent }
 6     states: [
 7         State {
 8             name: "pressed"
 9             when: mouse.pressed
10             PropertyChanges { target: rect; color: "red" }
11         }
12     ]
13 }
```

Listagem 9.12. Utilização da propriedade when do elemento State para modificação de estado de um objeto.

9.4.2. Animação com com elemento Behavior

Em QML, a transição de um estado para outro ocorre de maneira brusca e como consequência os valores das propriedades também são alterados bruscamente. Porém, muitas vezes, o esperado é que a transição ocorra de maneira suave, ou seja, através de animações, que são criadas através da interpolação entre os valores inicial e final de uma propriedade.

O elemento Behavior é utilizado com o propósito de se criar animações em QML. Esse elemento define uma animação padrão que deve ser utilizada sempre que uma de-

terminada propriedade de um objeto sofrer alterações. Uma boa prática de programação para esse elemento é utilizar estados bem definidos para realizar as transições, ou seja, evitar a utilização do estado base (“”). Essa prática é recomendada porque se uma animação for interrompida, o estado base assume os valores intermediários das propriedades e o comportamento final pode não ser o esperado.

A Listagem 9.13 apresenta um exemplo da utilização do elemento Behavior. Nesse exemplo, foram declarados dois estados no objeto rect, denominados “EstadoInicial” e “EstadoFinal”. A propriedade x possui o valor 0 no primeiro estado e o valor 400 no segundo estado. A declaração Behavior on x é responsável por realizar a interpolação dos valores dessa propriedade entre os dois estados que foram declarados. A interpolação é do tipo NumberAnimation e deve ter uma duração de 200ms.

Além do tipo NumberAnimation existem outras animações baseadas nos tipos de dados. Algumas das animações mais utilizadas são: ColorAnimation, para animar mudanças nos valores de cor; RotationAnimation, para animar rotações; e PropertyAnimation, para animar mudanças nos valores de propriedades.

```
 1 import QtQuick 1.0
 2
 3 Rectangle {
 4     width: 500; height: 100
 5     MouseArea { id: mouse; anchors.fill: parent }
 6     Rectangle {
 7         id: rect; width: 100; height: 100; color: "blue"
 8         Behavior on x { NumberAnimation{duration: 200} }
 9         states: [
10             State {
11                 name: "EstadoInicial"
12                 when: !mouse.pressed
13                 PropertyChanges { target: rect; x: 0 }
14             },
15             State {
16                 name: "EstadoFinal"
17                 when: mouse.pressed
18                 PropertyChanges { target: rect; x: 400 }
19             }
20         ]
21     }
22 }
```

Listagem 9.13. Exemplo de animação utilizando o elemento Behavior.

Uma propriedade pode ter somente um elemento Behavior associado à ela. Para realizar múltiplas transições com diferentes propriedades é necessário utilizar os elementos ParallelAnimation ou SequentialAnimation, para animações paralelas ou sequenciais, respectivamente.

9.4.3. Transições entre estados

Outra forma de se fazer animações em transições entre estados é utilizando o elemento Transition. Esse elemento define as animações que devem acontecer quando ocorrem

trocas de estados. As transições são atribuídas a um objeto através da propriedade transitions, que pode receber um único objeto ou uma lista de objetos do tipo Transition.

O elemento Transition possui 4 propriedades: animation, que é um lista de animações que devem ser executadas durante a transição; from e to, indicam que a transição é aplicável somente quando ocorrer a mudança de estados especificada nessas propriedades; e reversible, que indica se a animação da transição deve ser automaticamente revertida quando as condições de ativação forem invertidas. Essa última propriedade evita a criação de uma nova transição para realizar a animação reversa quando as condições de ativação estiverem invertidas.

As transições podem conter elementos de animação para interpolar as mudanças causadas por mudanças de estados nos valores das propriedades. Os elementos utilizados podem ser os mesmos utilizados com o elemento Behavior apresentado na Seção 9.4.2. Porém, se uma mudança de estado altera uma propriedade que foi declarada tanto em um Transition quanto em um Behavior, o comportamento definido no elemento Transition sobrepõe o elemento Behavior.

A Listagem 9.14 apresenta um exemplo da utilização do elemento Transition para realizar animações. Nesse exemplo, o elemento define animações para as propriedades color e x do objeto obj utilizando os elementos de animação ColorAnimation e NumberAnimation, respectivamente.

```
 1 import QtQuick 1.0
 2
 3 Rectangle {
 4      width: 200; height: 100
 5      Rectangle { id: obj; width: 100; height: 100; color: "red" }
 6      MouseArea { id: mouse; anchors.fill: parent }
 7      states: State {
 8          name: "EstadoFinal";
 9          when: mouse.pressed
10          PropertyChanges { target: obj; x: 100; color: "green" }
11      }
12      transitions: Transition {
13          NumberAnimation { properties: "x"; duration: 200 }
14          ColorAnimation { duration: 200 }
15      }
16  }
```

Listagem 9.14. Exemplo da utilização do elemento Transition para realizar animações.

9.5. Componentes QML Customizados

A linguagem QML é fortemente baseada nos conceitos de componentização. Dessa maneira, qualquer trecho de código pode ser tornar um componente e a própria declaração de elementos pode ser vista como o uso de componentes pré-definidos.

9.5.1. Definição de Componentes

Um componente QML é como uma caixa preta que interage com o mundo externo através de suas propriedades, sinais e funções, e geralmente é definido dentro de um arquivo

QML. Um ponto importante a ser considerado é que arquivos QML que definem componentes deve ter seus nomes iniciados com letra maiúscula. A Listagem 9.15 apresenta um exemplo de um componente Button, definido em um arquivo *Button.qml*, que emite sinais à medida que recebe cliques do usuário.

```
 1 // Arquivo: Button.qml
 2 import QtQuick 1.0
 3
 4 Rectangle {
 5     id: button
 6     anchors.centerIn: parent
 7     width: 100; height: 100
 8     radius: 50; color: 'red'
 9     signal clicked
10     MouseArea {
11         anchors.fill: button
12         onClicked: {
13             button.clicked()
14         }
15     }
16 }
```

Listagem 9.15. Componente customizado criado com o nome de arquivo *Button.qml*.

Componentes em QML também podem ser criados de uma maneira mais explícita usando o elemento Component, que permite uma definição *inline*, ou seja, dentro de um documento QML, ao invés de um arquivo separado. A vantagem do uso desse elemento é permitir reusar pequenas definições de componente dentro de um arquivo QML, ou para definir componentes que pertencem logicamente a um mesmo arquivo QML. Assim, o elemento Component gera componentes abstratos que servem como um modelo para reproduzir e renderizar cópias do componente definido. É importante esclarecer que o elemento Component apenas atua na definição de componentes e essa definição pode ser utilizada por outros elementos e métodos, tais como:

- **Loader**. É um elemento usado para carregar componentes QML visuais dinamicamente. Através dele é possível especificar como fonte de referência tanto um arquivo QML, usando a propriedade source, quanto um objeto Component, este último utilizando a propriedade sourceComponent;
- **createObject**. É um método pertencente aos objetos do tipo Component, que permite criar dinamicamente objetos a partir de suas definições de componente. Este método recebe como parâmetro a referência (id) do objeto que será o pai do novo objeto criado.

A Listagem 9.16 apresenta um exemplo da definição de um componente como um círculo de cor verde com id igual a circleComponent. Nesse exemplo, o elemento Loader é declarado, especificando o objeto circleComponent como sendo o componente de referência, através da propriedade sourceComponent, e então um círculo verde é renderizado na tela do dispositivo na posição padrão (x = 0 e y = 0).

O mesmo exemplo mostra a declaração de um elemento MouseArea, onde é declarado um *slot* onMousePositionChanged que é chamado no momento em que ocorre uma mudança no ponto de clique na tela, ou seja, no momento em que a área de toque é arrastada. Nesse *slot* é listada uma sequência de comandos visando a criação de objetos dinamicamente segundo a definição feita no componente circleComponent. A cada vez que o *slot* onMousePositionChanged é chamado, o método createObject() de circleComponent cria um novo objeto do mesmo tipo dentro do objeto mainView. Quando ocorre a criação do círculo dentro do *slot*, algumas das suas propriedades são alteradas diretamente no objeto armazenado na variável JavaScript circle. Assim, o novo círculo passa a ter seu centro no posição atual do mouse, e sua cor passa a ser azul.

```
 1 import QtQuick 1.0
 2
 3 Item {
 4     id: mainView; width: 360; height: 640
 5     Component {
 6         id: circleComponent
 7         Rectangle {
 8             id: circle
 9             width: 40; height: 40; radius: 40; color: "green"
10         }
11     }
12     Loader { id: pointer; sourceComponent: circleComponent}
13     MouseArea {
14         anchors.fill: mainView
15         onMousePositionChanged: {
16             var circle = circleComponent.createObject(mainView)
17             circle.x = mouseX - circle.width/2
18             circle.y = mouseY - circle.height/2
19             circle.color = 'blue'
20         }
21     }
22 }
```

Listagem 9.16. Definição e carregamento de um componente para desenhar um círculo.

9.5.2. Criação Dinâmica de Componentes com JavaScript

A utilização de código JavaScript é outra maneira de especificar os componentes. Essa abordagem bastante usada no desenvolvidos de jogos em QML, onde é preciso criar um número grande de objetos referentes à cena de jogo e fazer o gerenciamento destes à medida em que ocorrem alterações no estado do jogo.

Para definir componentes dentro de um trecho de código JavaScript é utilizada a função createComponent() pertencente a variável global Qt. Esta função requisita como parâmetro a url do arquivo QML onde o componente foi declarado e retorna um objeto do tipo Component.

A criação de objetos a partir de componentes que foram definidos é possível por meio do método createObject(). Esta função pertence a objetos do tipo Component, e recebe como parâmetros a referência do item que será o pai do novo objeto e um valor do tipo dicionário, uma série de pares chave-valor com as configurações das propriedades. O

retorno desta função é um objeto definido da forma como é declarado no Component e com suas propriedades alteradas segundos os valores passados como argumento. Assim é possível criar diferentes objetos de uma definição de componente e customizar livremente seus valores de propriedades.

O código apresentado na Listagem 9.17 demonstra a criação de um componente do tipo Button, mostrando que o objeto criado segue a mesma interface definida na declaração inicial do componente. Um objeto do tipo Button é criado de maneira dinâmica através do método createComponent() e é armazenado na variável buttonComponent. A variável buttonObj, por sua vez, recebe um objeto Button após a chamada do método createObject() do objeto buttonComponent, conforme apresentado na linha 14.

```
 1 import QtQuick 1.0
 2
 3 Rectangle {
 4     id: mainView
 5     width: 360
 6     height: 640
 7
 8     function randomColor() {
 9         return Qt.rgba(Math.random(),
10                        Math.random(),
11                        Math.random(),
12                        0.5 + (Math.random())/2)
13     }
14
15     Component.onCompleted: {
16         var buttonComponent = Qt.createComponent("Button.qml");
17         var buttonObj = buttonComponent.createObject(
18                 mainView,
19                 {"color" : randomColor()});
20         buttonObj.clicked.connect(function() {
21                                       color = randomColor()
22                                   });
23     }
24 }
```

Listagem 9.17. Criação de componentes utilizando código JavaScript.

Tanto as propriedades quando os sinais fazem parte da interface de um componente QML, assim todos os objetos criados a partir de um componente possuem as mesmas definições de propriedades e sinais. Logo, ainda na Listagem 9.17, é possível conectar um *slot* ao sinal clicked do objeto buttonObj do tipo Button (linha 16), onde tal sinal foi definido. O papel do *slot* conectado ao sinal clicked é atribuir um valor de cor aleatório a propriedade color à tela de fundo do aplicativo. É interessante perceber que apesar de estar dentro de um *slot* do objeto do tipo Button, o escopo de propriedade pertence a mainView, dado que o Component.onCompleted é declarado em mainView. Decorre então que a propriedade color refere-se ao objeto mainView e não ao próprio Button. Como resultado, tem-se um botão centralizado no objeto pai, mainView, e que ao ser pressionado altera a cor de fundo da tela.

9.5.2.1. Criação de Objetos a partir de uma String

Uma maneira interessante de criação de objetos QML em tempo de execução é a partir de uma string utilizando código JavaScript. Esses objetos são criados através do método createQmlObject() disponível na variável global Qt, conforme apresentado na Listagem 9.18.

```
1 var newObj = Qt.createQmlObject(
2     'import QtQuick 1.0;' +
3     'Rectangle { color: "red"; width: 20; height: 20}',
4     parentItem,
5     "dynamicSnippet1");
```

Listagem 9.18. Novo objeto criado a partir de uma string utilizando código JavaScript

O primeiro argumento do método define o componente, utilizando o código com os mesmos elementos de declarações em arquivos QML. Se o código definido na string possuir algum caminho de arquivo, este deve ser relativo ao item pai do objeto criado. O segundo argumento é o item pai do objeto a ser criado. Por fim, o terceiro argumento é apenas um nome de diretório associado ao novo objeto, caso seja necessário reportar erros.

9.5.2.2. Deletando Objetos Dinamicamente

De maneira geral, uma grande quantidade de itens criados dinamicamente poder trazer impactos negativos ao desempenho de um aplicativo. Assim, é altamente recomendável que se tenha um bom gerenciamento dos objetos criados. Deletar os objetos criados dinamicamente à medida que se tornarem desnecessários pode beneficiar o desempenho como um todo.

É importante pontuar a diferença entre itens criados dinamicamente por funções JavaScript e aqueles criados por elementos QML, tais como o Loader. Estes últimos possuem em sua implementação o pressuposto de que seus itens gerados não serão deletados durante o tempo de vida do componente que os engloba. Assim deve ser evitado deletar qualquer item que não tenha sido diretamente criado por funções JavaScript.

Todo item possui um método de destruição chamado destroy(). Este método possui um argumento opcional onde pode ser passado o tempo em milissegundos antes que o objeto seja destruído. A Listagem 9.19 apresenta um exemplo de criação dinâmica através de uma string e destruição do objeto criado. O trecho de código definido no *slot* onCompleted é executado no momento em que o item pai mainView é carregado e renderizado na tela. Então, um novo objeto é criado (um retângulo vermelho de bordas arredondadas) cujo pai é definido como sendo o mainView e seu caminho para relatórios de erros como "snippetCode1". A variável newObject passa a armazenar o objeto retângulo, enquanto na linha seguinte uma variável newMouseArea recebe uma instância do tipo MouseArea. Ao final do exemplo, um *slot* é conectado ao sinal clicked do MouseArea, onde o primeiro objeto tem sua opacidade modificada para zero e então seu método de destruição é chamado.

```
 1 import QtQuick 1.0
 2
 3 Rectangle {
 4     id: mainView; width: 640; height: 360
 5     Component.onCompleted: {
 6         var newObject = Qt.createQmlObject(
 7            " import QtQuick 1.0; " +
 8            " Rectangle { anchors.centerIn: parent; " +
 9            "             color: 'red'; width: 200; height: 200; " +
10            "             radius: 20}",
11            mainView, "snippetCode1");
12         var newMouseArea = Qt.createQmlObject(
13            " import QtQuick 1.0; " +
14            " MouseArea { anchors.fill: parent;}",
15            newObject, "mouseArea1");
16         newMouseArea.clicked.connect(function() {
17                         newObject.opacity = 0;
18                         newObject.destroy(1000)
19                 });
20      }
21 }
```

Listagem 9.19. Exemplo da utilização do método destroy() para a destruição de objetos criados dinamicamente.

9.5.3. Definição de Componentes utilizando Qt/C++

A grande diversidade de elementos QML permitem criar variados tipos de interface de maneira declarativa. Entretanto para aplicações que exijam renderizações de forma geométricas complexas ou possuam comportamento funcional acima do disponibilizado pelos elementos QML padrões, existe a possibilidade de estender novas funcionalidade através de classes C++. Dessa maneira, elementos QML com alto grau de customização podem ser definidos em uma classe C++ que herde o QDeclarativeItem, a classe pai de todos os itens declarativos.

Feita a sobrecarga do método de paint do QDeclarativeItem, é possível criar um componente QML utilizando todas as potencialidades oferecidas pelo framework Qt. Na Listagem 9.20, um triângulo é desenhado através de um caminho de pontos pré-definidos, um QPaintPath. O painter é configurado para realizar a renderização do caminho de pontos definido, através do método drawPath do QPainter. Os métodos setPen e setBrush apenas selecionam a cor preta para os traços do desenho e uma cor pré-definida m_color para o preenchimento da figura.

9.5.3.1. Criação de Novos Tipos

Uma vez que um novo elemento QML é criado em C++, é necessário registrá-lo para uso dentro de arquivos QML, ou seja, é preciso certificar que a máquina de execução QML consiga compreender o significado dos novos símbolos definidos.

Especificada no cabeçalho qdeclarativeengine.h, a função qmlRegisterType re-

```
 1 class MyItem : public QDeclarativeItem
 2 {
 3    ...
 4    void paint(QPainter *painter,
 5      const QStyleOptionGraphicsItem *,
 6      QWidget *) {
 7        QPainterPath path;
 8        path.moveTo(boundingRect().width()/2,0);
 9        path.lineTo(boundingRect().width(),boundingRect().height());
10        path.lineTo(0,boundingRect().height());
11        path.lineTo(boundingRect().width()/2,0);
12
13        painter->setPen(QColor("black"));
14        painter->setBrush(QBrush(m_color));
15        painter->drawPath(path);
16    }
17 };
```

Listagem 9.20. Definição de um novo componente utilizando código Qt/C++.

gistra os tipos C++ no sistema QML, através da especificação de um URI (*Uniform Resource Identifier*) da biblioteca, um número de versão e o nome do componente. Abaixo segue um exemplo de seu uso dentro do código C++, onde o elemento MyItem é declarado através do registro da classe MyItem dentro da biblioteca MyComponents sob a versão 1.0.

```
qmlRegisterType<MyItem>("MyComponents", 1, 0, "MyItem");
```

O item customizado pode ser facilmente importado para o código QML, sendo que para isso é necessário conhecer o nome e versão da biblioteca onde está agrupado, assim como seu nome de componente. Uma vez importado, o novo elemento segue exatamente as mesmas regras de qualquer outro elemento do tipo Item, possuindo assim também suas propriedades, tais como x, y, width, height e outras. A Listagem 9.21 apresenta um exemplo da utilização do elemento do novo tipo (MyItem) criado a partir de código C++.

```
1 import MyComponents 1.0
2
3 Rectangle {
4     MyItem {
5         id: myItem
6     }
7 }
```

Listagem 9.21. Declaração do novo elemento criado a partir de código C++.

9.6. Plataformas de Desenvolvimento

No contexto de dispositivos móveis, Symbian e Maemo/MeeGo são algumas plataformas que oferecem suporte ao desenvolvimento de aplicações em Qt. No desenvolvimento em Qt, geralmente não são necessárias muitas modificações no código para que um projeto

existente funcione em uma nova plataforma. Dessa maneira, na maioria dos casos o processo de compilar o código para essa nova plataforma e executar a aplicação funciona sem maiores problemas. Porém, algumas vezes é necessário realizar configurações de projeto no arquivo *.pro* para que o aplicativo tenha acesso aos recursos do dispositivo.

A Listagem 9.22 apresenta algumas configurações específicas da plataforma Symbian utilizadas no desenvolvimento de aplicações. As duas instruções apresentadas nessa listagem são: TARGET.CAPABILITIES, que define privilégios extras para que a aplicação tenha acesso aos recursos do dispositivo, como é o caso dos serviços de rede com NetworkServices; e TARGET.EPOCHEAPSIZE, que define a quantidade mínima e máxima de memória disponível para a aplicação.

```
1 symbian: {
2     ...
3     TARGET.CAPABILITY += NetworkServices
4     TARGET.EPOCHEAPSIZE = 0x020000 0x4000000
5     ...
6 }
```

Listagem 9.22. Configurações específicas da plataforma Symbian para desenvolvimento Qt.

Na plataforma Maemo/MeeGo, a IDE Qt Creator cria os projetos com todos os recursos necessários para o desenvolvimento. No caso da plataforma Android, ainda não existe um *port* oficial, mas já existem algumas iniciativas para utilizar desenvolver com o Framework Qt nessa plataforma. O Necessitas [3] é um projeto que tem o objetivo de prover o *port* do Qt para a plataforma Android e integração com o Qt Creator. O site do projeto fornece as informações necessárias para começar a utilizar o Necessitas SDK.

Quanto ao processo de compilação das aplicações para instalação no dispositivo móvel, o QtSDK fornece o módulo *Remote Compiler*. Esse módulo foi criado para contornar a ausência de soluções que permitissem a compilação dos binários Symbian nas plataformas Linux e Apple Mac. Esse recurso permite a compilação para as plataformas Symbian S60, Symbianˆ1, Symbianˆ3, Maemo 5 e MeeGo 1.2 Harmattan, oferecendo suporte para a versão 4.6 do Qt, incluindo o Qt Quick a partir da versão 4.7 do Qt.

9.7. Considerações Finais

Os elementos e características da linguagem QML apresentados neste capítulo possibilitam o desenvolvimento *cross-platform* de UIs de alto nível de maneira rápida e intuitiva. Para reforçar a importância desse kit de desenvolvimento, o framework Qt vem sendo utilizado por grandes empresas em projetos como Google Earth, Skype, KDE e Opera [7]. O Ambiente declarativo Qt Quick, através da linguagem QML, surge como uma ferramenta promissora para a criação de UIs onde a rapidez de desenvolvimento e a experiência do usuário são cruciais para as aplicações. Além disso, novos dispositivos móveis recentemente lançados com o sistema operacional MeeGo possuem o Qt como kit de desenvolvimento padrão.

Em resumo, neste capítulo foi apresentado como utilizar a linguagem QML no ambiente declarativo Qt Quick para a criação de interfaces para dispositivos móveis. Através de elementos básicos como Rectangle, Item, MouseArea, Text e Image foi apresen-

tado como especificar UIs pelo seu conteúdo, e não por meio de comandos imperativos em Qt/C++ na especificação dos métodos. Além disso, outras características importantes da linguagem QML (como a criação de animações e o tratamento de eventos) foram apresentadas, possibilitando a criação de UIs modernas e flexíveis.

9.8. Agradecimentos

A escrita deste capítulo foi apoiada pelo CETELI (Centro de Pesquisa e Desenvolvimento em Tecnologia Eletrônica e da Informação) e pelo INdT (Instituto Nokia de Tecnologia).

Referências

[1] Goderis, S. (2005) "High-level declarative user interfaces". In: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05). New York, NY, USA, ACM, p. 236-237.

[2] QUIt Coding, Crash course to Qt Quick Game Programming, disponível em: http://quitcoding.com/download/Qt_Quick_Game_Programming_1_0.pdf, acessado em outubro de 2011.

[3] Necessitas, disponível em: http://sourceforge.net/p/necessitas/home/necessitas/, acessado em outubro de 2011.

[4] Nokia, QML Elements, disponível em: http://doc.qt.nokia.com/4.7-snapshot/qdeclarativeelements.html, acessado em outubro de 2011.

[5] Nokia, Qt - Cross-platform application and UI framework, disponível em: http://qt.nokia.com/, acessado em outubro de 2011.

[6] Nokia, Qt 4.7: QML Viewer, disponível em: http://doc.qt.nokia.com/4.7-snapshot/qmlviewer.html, acessado em outubro de 2011.

[7] Nokia, Qt in Use, disponível em: http://qt.nokia.com/qt-in-use/, acessado em outubro de 2011.

[8] Nokia, Qt Quick, http://qt.nokia.com/qtquick/, acessado em outubro de 2011.

[9] Vision Mobile, Developer Economics 2011, Disponível em: http://www.visionmobile.com/devecon.php, acessado em outubro de 2011.

[10] Vision Mobile, Mobile Economics Developer 2010 and Beyond, Disponível em: http://www.visionmobile.com/rsc/researchreports/Mobile%20Developer %20Economics%202010%20Report%20FINAL.pdf, acessado em outubro de 2011.

[11] W3C Recommendation, Basic Data Types and Interfaces, disponível em: http://www.w3.org/TR/SVG/types.html#ColorKeywords, acessado em outubro de 2011.

[12] Zucker, D. e Rischpater, R., Beginning Nokia Apps Development: Qt and HTML5 for Symbian and MeeGo, Apress Series. Apress, 2010.