CENTRO EDUCACIONAL DA FUNDAÇÃO SALVADOR ARENA

FACULDADE DE TECNOLOGIA TERMOMECANICA

MATHEUS SUAREZ SILVA

RICARDO CARDOSO PETRÉRE

DESENVOLVIMENTO DE UMA APLICAÇÃO

MULTIPLATAFORMA UTILIZANDO QT

SÃO BERNARDO DO CAMPO

2014

MATHEUS SUAREZ SILVA

RICARDO CARDOSO PETRÉRE

DESENVOLVIMENTO DE UMA APLICAÇÃO

MULTIPLATAFORMA UTILIZANDO QT

Trabalho de Conclusão de Curso, realizado sob orientação do Prof. Ms. Eduardo Rosalém Marcelino, apresentado à Faculdade de Tecnologia Termomecanica como requisito para obtenção do título de Tecnólogo.

SÃO BERNARDO DO CAMPO

2014

MATHEUS SUAREZ SILVA

RICARDO CARDOSO PETRÉRE

DESENVOLVIMENTO DE UMA APLICAÇÃO

MULTIPLATAFORMA UTILIZANDO QT

Trabalho de Conclusão de Curso – Faculdade de Tecnologia Termomecanica

Comissão Julgadora

Professor Ms. Eduardo Rosalém Marcelino

Professor Ms. Ricardo S. Jacomini

Joel Henrique Silva

SÃO BERNARDO DO CAMPO

2014

**AGRADECIMENTOS**

Prestamos nossos sinceros agradecimentos a todos aqueles que tornaram possível esse momento de encerramento de mais um ciclo de nossas vidas.

Agradecemos primeiramente a Deus, que nos guiou com sua sabedoria infinita através desses três anos, nos permitindo concluir com sucesso essa fase.

A Fundação Salvador Arena, assim como todo o conselho curador, que nos possibilitaram o ingresso na Faculdade de Tecnologia Termomecanica e nos suportaram ao longo do curso com o que era necessário para que pudéssemos concluir com sucesso essa importante etapa.

Ao nosso orientador, professor Ms. Eduardo Rosalém Marcelino, que com seus conselhos, sugestões e paciência nos mostrou os melhores caminhos a trilhar para a conclusão desse trabalho.

A todo o corpo docente, que nos fez evoluir com a transmissão de seus conhecimentos, não somente na questão acadêmica, mas também para que pudéssemos criar uma nova perspectiva profissionalmente.

As nossas famílias, que nos apoiaram em nossa escolha e nos incentivaram sempre a continuar e não desistir nos momentos de fraqueza.

A todos nossos amigos, pela ajuda nos momentos difíceis, que de certa maneira nos ajudaram a passar por esse período de forma mais agradável e proporcionando vários momentos dos quais nos lembraremos com certeza pelo resto de nossas vidas.

*“O único lugar onde sucesso*

*vem antes de trabalho*

*é no dicionário”*

*Albert Einstein*

*ou*

*“O insucesso é apenas uma*

*oportunidade para recomeçar*

*de novo com mais inteligência”*

*Henry Ford*

**RESUMO**

Este trabalho busca reunir informações sobre desenvolvimento multiplataforma, assim como fazer uma breve abordagem sobre a história da computação. Nesta abordagem histórica observou-se que com o passar dos anos, o progresso tecnológico possibilitou a criação máquinas e sistemas operacionais cada vez mais robustos e diversos. A partir disso, foram criadas novas linguagens de programação que tornavam o desenvolvimento mais simples e que apresentavam várias funcionalidades que não poderiam existir antes devido aos limites tecnológicos. Porém, a diversidade de arquiteturas e sistemas operacionais trouxe um problema, pois cada plataforma interpreta códigos de maneiras distintas, fazendo com que um *software* desenvolvido para um sistema operacional funcione com falhas ou nao funcione nos demais. Na busca da solução para esse problema, algumas empresas criaram ferramentas ou *frameworks* que possibilitam o desenvolvimento de *softwares* que rodem de maneira similar em qualquer plataforma desejada, com poucas ou nenhuma mudança no código fonte. Esse trabalho utiliza o *framework* Qt para o desenvolvimento de um protótipo de troca de mensagens, para demonstrar o comportamento dele em plataformas distintas.

**Palavras-chaves:**Multiplataforma. Qt. Troca de Mensagens

**ABSTRACT**

This piece of work has the intent to gather information about cross-platform developing, and also make a historic approach about computation. In this historic approach, it was observed that as the years passed by, the technological progress made possible the creation of more robust and diverse machines and operating systems. Thereafter, new programming languages have been created that made developing simpler and that presented many new features that could not exist before due to technological limits. However, the diversity of architectures and operating systems has brought a problem, because each platform interprets code in different ways, causing a software developed for an operating system to work with bugs or not work at all in the others. In the pursuit of the solution for this problem, some companies have created tools or frameworks that made possible the development of softwares that work similarly on any desired platform, with few or no changes in the source code. This work uses the Qt framework for the development of a prototype of messages exchange, to demonstrate its behavior in distinct platforms.

**Keywords:**Cross-platform, Qt. Message exchange

**LISTA DE QUADROS**

[Quadro 1 – Métodos "Deserializar" e "Serializar" da classe EContato 69](#_Toc404228099)

[Quadro 2 – Método enviaPacote da classe ComunicacaoRede 70](#_Toc404228100)

**LISTA DE FIGURAS**

[Figura 1 – Logo do Qt 18](#_Toc404227660)

[Figura 2 – Programas que utilizam Qt 22](#_Toc404227661)

[Figura 3 – Exemplo de classe com *signals* e *slots* implementados 23](#_Toc404227662)

[Figura 4 – Implementação de emissão de *signal* 24](#_Toc404227663)

[Figura 5 – Exemplo do mecanismo de *signal* e *slot* 25](#_Toc404227664)

[Figura 6 – Exemplo de código-fonte com conexão entre objetos utilizando *signals* e *slots* 26](#_Toc404227665)

[Figura 7 – Wireshark 2 Preview em funcionamento 28](#_Toc404227666)

[Figura 8 – A comunidade KDE 29](#_Toc404227667)

[Figura 9 – Interface do Plasma 29](#_Toc404227668)

[Figura 10 – Tela inicial do Qt Creator 31](#_Toc404227669)

[Figura 11 – Exemplo de propriedades atreladas à outras (*property bindings*) 33](#_Toc404227670)

[Figura 12 – Exemplo de código em QML para criação de um retângulo 33](#_Toc404227671)

[Figura 13 – Alguns tipos utilizados no sistema de tipagem da linguagem QML. 34](#_Toc404227672)

[Figura 14 – Exemplo de tela com *widgets* aninhados 35](#_Toc404227673)

[Figura 15 – Exemplo de Estilos dos *Widgets*. 36](#_Toc404227674)

[Figura 16 – Exemplos de gerenciadores de *layouts* do módulo Qt Widgets 37](#_Toc404227675)

[Figura 17 – Logo da empresa Xamarin 38](#_Toc404227676)

[Figura 18 – Código para a criação de uma tela de *login* comum 40](#_Toc404227677)

[Figura 19 – Código mostrado na Figura 18 rodando, respectivamente, em iOS, Android e Windows Phone 40](#_Toc404227678)

[Figura 20 – Criação de interface para iOS utilizando-se o Xamarin para Visual Studio 42](#_Toc404227679)

[Figura 21 – Criação de interface para iOS utilizando-se o Xamarin Studio 43](#_Toc404227680)

[Figura 22 – Software Rdio rodando em iOS (tablet) e Android (smartphone) 44](#_Toc404227681)

[Figura 23 – Aplicativo da empresa Kimberly-Clark desenvolvido com utilização do Xamarin 45](#_Toc404227682)

[Figura 24 – Interface do Xamarin Test Cloud 47](#_Toc404227683)

[Figura 25 – Logotipo da Mono 47](#_Toc404227684)

[Figura 26 – Plataformas e arquiteturas suportadas 49](#_Toc404227685)

[Figura 27 – Tela principal do MonoDevelop 51](#_Toc404227686)

[Figura 28 – Tela de criação de interfaces da IDE MonoDevelop 52](#_Toc404227687)

[Figura 29 – Logotipo do Java 53](#_Toc404227688)

[Figura 30 – Logotipo da empresa Oracle 54](#_Toc404227689)

[Figura 31 – Processo de compilação em linguagem C 55](#_Toc404227690)

[Figura 32 – Comunicação do código binário diretamente com o sistema operacional 55](#_Toc404227691)

[Figura 33 – Processo de comunicação entre aplicação e sistema operacional através da JVM. 56](#_Toc404227692)

[Figura 34 – Exemplo de código escrito em Java 56](#_Toc404227693)

[Figura 35 – Bytecode gerado após a compilação do código mostrado na 57](#_Toc404227694)

[Figura 36 – Ambiente de desenvolvimento da Eclipse 58](#_Toc404227695)

[Figura 37 – Ambiente de desenvolvimento da NetBeans 58](#_Toc404227696)

[Figura 38 – Utilização de diferentes *look-and-feels* para uma mesma aplicação 59](#_Toc404227697)

[Figura 39 – Modelo de Arquitetura Cliente-Servidor da aplicação 60](#_Toc404227698)

[Figura 40 – Exemplo de interface de uma aplicação *desktop* 62](#_Toc404227699)

[Figura 41 – Protótipo da tela de mensagens na plataforma Android e Windows 63](#_Toc404227700)

[Figura 42 – Exemplo de comportamento ocorrido ao utilizar *dialog* em ambiente Android 64](#_Toc404227701)

[Figura 43 – Exemplo de caixa de diálogo de abertura de arquivo no Android 65](#_Toc404227702)

[Figura 44 – Classes contidas na pasta "vo" 66](#_Toc404227703)

**LISTA DE SIGLAS**

**AOT –** Ahead-of-Time

**API –** Application Programming Interface

**AWT –** Abstract Window Toolkit

**CSS** **–** Cascading Style Sheets

**DOS –** Disk Operating System

**ENIAC –** Electronic Numerical Integrator Analyzer and Computer

**GTK+ –** GIMP Toolkit

**GUI –** Graphical User Interface

**IDE –** Integrated Development Environment

**JDK –** Java Development Kit

**JIT –** Just-In-Time

**JRE –** Java Runtime Environment

**JVM –** Java Virtual Machine

**KDE –** K Desktop Environment

**MVC –** Model-View-Controller

**OP Code –** Código Operacional

**PDA** **–** Personal Digital Assistant

**PDF –** Portable Document Format

**QML –** Qt Meta-Objects Language

**QPL –** Q Public License

**SO –** Sistema Operacional

**SVG –** Scalable Vector Graphics

**SWT –** Standard Widget Toolkit

**UI –** User Interface

**WinRT –** Windows Runtime

**XML –** eXtended Markup Language

**SUMÁRIO**

[1 INTRODUÇÃO 14](#_Toc404227704)

[1.1 OBJETIVO GERAL 16](#_Toc404227705)

[1.2 OBJETIVOS ESPECÍFICOS 16](#_Toc404227706)

[1.3 JUSTIFICATIVA 16](#_Toc404227707)

[1.4 METODOLOGIA 17](#_Toc404227708)

[1.5 ESTRUTURA DO TRABALHO 17](#_Toc404227709)

[2 TECNOLOGIA UTILIZADA 18](#_Toc404227710)

[2.1 INTRODUÇÃO 18](#_Toc404227711)

[2.2 QT 18](#_Toc404227712)

[2.2.1 HISTÓRIA DO QT 18](#_Toc404227713)

[2.2.1.1 QT 4 20](#_Toc404227714)

[2.2.1.2 QT 5 21](#_Toc404227715)

[2.2.2 SINAIS E SLOTS (*SIGNALS AND SLOTS*) 22](#_Toc404227716)

[2.3 EXEMPLOS DE APLICAÇÕES QUE UTILIZAM QT 27](#_Toc404227717)

[2.3.1 WIRESHARK 27](#_Toc404227718)

[2.3.2 KDE 28](#_Toc404227719)

[2.4 QT CREATOR 31](#_Toc404227720)

[2.4.1 QT QUICK 32](#_Toc404227721)

[2.4.1.1 QML 32](#_Toc404227722)

[2.4.2 QT WIDGETS 34](#_Toc404227723)

[3 ALTERNATIVAS AO QT 38](#_Toc404227724)

[3.1 INTRODUÇÃO 38](#_Toc404227725)

[3.2 XAMARIN 38](#_Toc404227726)

[3.2.1 XAMARIN.FORMS 39](#_Toc404227727)

[3.2.2 IDEs 41](#_Toc404227728)

[3.2.3 CLIENTES DA XAMARIN 43](#_Toc404227729)

[3.2.4 XAMARIN TEST CLOUD 45](#_Toc404227730)

[3.3 MONO 47](#_Toc404227731)

[3.3.1 COMPONENTES DA MONO 47](#_Toc404227732)

[3.3.2 SISTEMAS OPERACIONAIS SUPORTADOS 48](#_Toc404227733)

[3.3.3 BENEFÍCIOS 49](#_Toc404227734)

[3.3.4 IDE 50](#_Toc404227735)

[3.4 JAVA 52](#_Toc404227736)

[3.4.1 HISTÓRIA DO JAVA 53](#_Toc404227737)

[3.4.2 COMPONENTES DO JAVA 54](#_Toc404227738)

[3.4.3 JAVA VITUAL MACHINE (JVM) 54](#_Toc404227739)

[3.4.4 IDEs 57](#_Toc404227740)

[3.4.5 INTERFACE GRÁFICA EM JAVA 59](#_Toc404227741)

[4 ESTUDO DE CASO 60](#_Toc404227742)

[4.1 INTRODUÇÃO 60](#_Toc404227743)

[4.2 TECNOLOGIAS UTILIZADAS 61](#_Toc404227744)

[4.2.1 SQL SERVER 61](#_Toc404227745)

[4.2.2 GITHUB 61](#_Toc404227746)

[4.3 APLICAÇÃO CLIENTE (QTCC) 61](#_Toc404227747)

[4.3.1 INTERFACE DA APLICAÇÃO 62](#_Toc404227748)

[4.3.2 SERIALIZAÇÃO EM JSON 65](#_Toc404227749)

[4.3.3 ENVIO PELA REDE 68](#_Toc404227750)

[4.4 APLICAÇÃO SERVIDOR (QTCC\_Server) 69](#_Toc404227751)

[4.4.1 INTERFACE DA APLICAÇÃO 69](#_Toc404227752)

[4.4.2 SERIALIZAÇÃO EM JSON 69](#_Toc404227753)

[4.4.3 ENVIO PELA REDE 70](#_Toc404227754)

[4.4.4 PERSISTÊNCIA DOS DADOS 70](#_Toc404227755)

[5 CONCLUSÃO 71](#_Toc404227756)

[6 TRABALHOS FUTUROS 72](#_Toc404227757)

[7 REFERÊNCIAS 73](#_Toc404227758)

1. INTRODUÇÃO

Vários anos se passaram desde a criação dos primeiros computadores, que basicamente faziam cálculos antes feitos manualmente.

As máquinas que precederam os computadores da forma como são hoje eram chamadas de tabuladoras, e elas eram capazes de processar dados através da separação de cartões perfurados. O funcionamento desse sistema era bastante simples: a máquina atribuía o valor 0 (zero) para um espaço sem furo e o valor 1 (um) para furado.

A máquina tida como o primeiro computador digital-eletrônico, o *Electronic Numerical Integrator Analyzer and Computer* (ENIAC) era programável manualmente, através do uso de fios e chaves. Os dados a serem processados entravam via cartão perfurado, e seus programas costumavam demorar de uma hora a um dia inteiro para serem criados e executados (FONSECA FILHO, 2007).

Com o avanço da tecnologia, os computadores foram ficando cada vez menores, e o sistema de cartões perfurados foi substituído pela proposta do matemático húngaro John Von Neumann, que sugeriu o armazenamento das instruções antes passadas através de cartões perfurados na memória do computador, o que tornaria o acesso à elas mais rápido. Essa proposta deu certo, e os computadores de hoje seguem esse mesmo modelo.

Com essa evolução, começaram a surgir as primeiras linguagens de programação de alto nível por volta da década de 1950, e elas requeriam um compilador, que interpretava o código escrito e gerava um equivalente em linguagem de máquina.

Os primeiros Sistemas Operacionais (S.O.) modernos surgiram entre a década de 1960 e 1970, sendo todos baseados em Unix, que foi o primeiro a ser escrito em linguagem C. Porém, ele se tratava de um sistema operacional para máquinas de grande porte, e com a popularização dos computadores pessoais, foi necessário o desenvolvimento de S.Os mais simples, sendo o primeiro deles o *Disk Operating System* (DOS), desenvolvido por Tim Paterson, e adquirido pelos fundadores da Microsoft, William Gates e Paul Allen. Este SO vendeu muitas cópias e foi considerado o sistema operacional padrão para computadores pessoais na década de 80 (FONSECA FILHO, 2007).

A partir disso, novos sistemas operacionais foram desenvolvidos, cada vez mais robustos, acompanhando o desenvolvimento dos microcomputadores, sendo os principais o Windows, desenvolvido e distribuído pela Microsoft, o Machintosh, cujo nome foi posteriormente mudado para Mac OS e OS X, desenvolvido e distribuído pela Apple em seus computadores, e o que viria a ser o Linux, desenvolvido por Linus Torvalds, que é um sistema operacional livre (gratuito).

Mais recentemente, foram criados sistemas operacionais para aparelhos *mobile*, os chamados smartphones.

Com o aumento na quantidade de sistemas operacionais, começou-se a surgir o problema de disponibilizar os mesmos programas para cada uma dessas plataformas. Cada SO possui características próprias, sejam elas a arquitetura interna, o sistema de arquivos, ou até mesmo o conjunto de bibliotecas de sistema utilizado. Portanto, cada sistema operacional só é capaz de executar específicos tipos de programas, tornando impossível o conceito de utilizar um mesmo arquivo executável em diferentes plataformas.

De forma a conseguir disponibilizar a mesma aplicação em diferentes sistemas, era necessário criar diferentes versões do mesmo *software*, basicamente reescrevendo boa parte do código-fonte da aplicação. Isso impacta em diversos aspectos do processo de desenvolvimento de *software*, pois aumenta o tempo gasto com uma mesma correção para cada versão do programa, maior mão-de-obra empregada na equipe, além de mais dificuldade em manter o *software* atualizado com as mais novas implementações e funcionalidades.

De modo a solucionar tal problema, foram criadas ferramentas capazes de gerar aplicações chamadas “multiplataforma[[1]](#footnote-1)”, e, dentre essas ferramentas, uma das mais utilizadas no mercado, além de uma das mais antigas e completas, é o *framework* Qt.

O presente trabalho abordará a utilização da linguagem de programação de alto nível C++ aplicada para desenvolvimento multiplataforma, através do *framework* de aplicações Qt e a IDE Qt Creator, que, utilizando compiladores específicos para cada plataforma, permite que o mesmo código-fonte funcione em vários sistemas operacionais sem a necessidade de adaptações, como em outras linguagens e IDEs.

* 1. OBJETIVO GERAL

Estudo da tecnologia Qt no que tange o desenvolvimento de aplicações multiplataforma, analisando sua viabilidade e necessidade ou não de refatoração de código.

* 1. OBJETIVOS ESPECÍFICOS

Para atingir o objetivo principal, os seguintes passos serão seguidos:

Revisão bibliográfica sobre as tecnologias abordadas nesse trabalho de conclusão de curso, incluindo desenvolvimento e compilação de aplicações para sistemas operacionais distintos.

Verificar se uma aplicação Qt desenvolvida para uma determinada plataforma pode ser executada em outras plataformas sem a necessidade ou com um mínimo de adaptações no código fonte necessárias.

Desenvolvimento de um protótipo de software de envio de mensagens instantâneas com o intuito de analisar o Qt, focando principalmente sua característica que permite o desenvolvimento de aplicações multiplataforma.

* 1. JUSTIFICATIVA

O desenvolvimento de aplicações e softwares pode parecer algo complicado, ainda mais quando o software desenvolvido tem como destino mais de um sistema operacional, pois cada SO interpreta o código de uma maneira diferente. Para facilitar a vida dos desenvolvedores, algumas ferramentas e linguagens se propõem a anular ou minimizar as mudanças necessárias em código para que as aplicações funcionem perfeitamente em mais de uma plataforma. Dentre essas ferramentas, há o Qt, que trata-se de uma ferramenta que se propõe a desenvolver aplicações gráficas, multiplataformas e com todos os recursos oferecidos pela linguagem C++ **bla bla bla...**  **Se encontrarem dados sobre o uso e crescimento do Qt, aqui é um bom lugar para escrever e justificar seu uso e estudo.**

* 1. METODOLOGIA

Levantamento bibliográfico. Estudo de Caso, através do desenvolvimento de um protótipo de aplicação para trocas de mensagens de texto entre clientes rodando em diferentes plataformas[[2]](#footnote-2).

* 1. ESTRUTURA DO TRABALHO

Para contextualizar o trabalho, o documento foi dividido da seguinte maneira:

* No capítulo 2, intitulado “Tecnologia utilizada”, serão abordados diversos aspectos sobre a programação multiplataforma, focando-se no *framework* de aplicações Qt e a ferramenta que o utiliza, o Qt Creator;
* No capítulo 3, sob o título “Alternativas ao Qt”, serão apresentadas ferramentas e linguagens de programação com o mesmo intuito do Qt, e será feita uma breve análise comparativa entre as principais alternativas, assim como a análise de vantagens e desvantagens entre elas;
* No capítulo 4, sob o tema “Estudo de caso”, estarão descritas as funcionalidades do protótipo de aplicação para múltiplas plataformas, assim como seu processo de desenvolvimento;
* No capítulo 5, denominado “Conclusão” será apresentada a conclusão do trabalho, com destaque para as experiências adquiridas e as lições aprendidas;
* No capítulo 6, intitulado “Trabalhos futuros”, serão apresentadas sugestões para aqueles que desejarem continuar esse trabalho.
* Por fim, no capítulo 7, sob o título “Referências”, serão apresentadas as referências bibliográficas em que foram baseadas as pesquisas feitas visando a conclusão deste trabalho.

1. TECNOLOGIA UTILIZADA
   1. INTRODUÇÃO

Nesse capítulo serão abordados os principais aspectos sobre o *framework* de desenvolvimento multiplataforma Qt e a ferramenta que utiliza esse *framework* como base, o Qt Creator. Serão abordados temas como história, plataformas suportadas, tipos de aplicações e vantagens em sua utilização.

* 1. QT

Segundo Blanchette e Summerfield (2008), o Qt é um *framework* de desenvolvimento em C++ multiplataforma utilizando a filosofia “escreva uma vez, compile em qualquer lugar”. Seu intuito é de que programadores possam desenvolver aplicações utilizando apenas um código-fonte e compilando-o para as diversas plataformas nas quais seu programa será utilizado, sem alterações no código. A Figura 1 ilustra o logo do Qt.



Figura 1 – Logo do Qt

Fonte: Qt Digia, 2014a

* + 1. HISTÓRIA DO QT

De acordo com Blanchette e Summerfield (2008), o projeto que se tornaria no futuro o *framework* Qt foi idealizado em 1990 por Haavard Nord e Eirik Chambe-Eng, ambos mestres em ciência da computação pelo *Norwegian Institute of Technology* (Instituto Norueguês de Tecnologia). A ideia surgiu devido à necessidade deles de desenvolver uma aplicação em C++ que possuísse uma interface gráfica em Unix, Machintosh e Windows. Para resolver o problema que possuíam, iniciaram o desenvolvimento de um *framework* multiplataforma que suportava o paradigma de orientação a objetos.

Em 1991, Haavard começou a escrever as primeiras classes do *framework* com a ajuda de Eirik, que era responsável pelo *design*. Em 1993, eles haviam desenvolvido o primeiro kernel gráfico do *framework* Qt e puderam implementar suas primeiras aplicações.

O nome Qt foi criado da seguinte forma: a letra Q foi escolhida como prefixo por sua aparência no editor de texto Emacs de Haavard, enquanto que a letra t vem de *toolkit* (kit de ferramentas), inspirado pelo Xt, uma ferramenta para o desenvolvimento de aplicações *Graphical User Interface* (GUI) para o sistema X11, comum em algumas distribuições Unix (BLANCHETTE; SUMMERFIELD, 2008).

Em 1995, o Qt 0.90 foi liberado ao público pela primeira vez. O novo *framework* podia ser utilizado para o desenvolvimento tanto em Windows quanto em Unix, oferecendo a mesma *Application Programming Interface* (API)para ambas as plataformas. O Qt estava disponível em dois tipos de licenças: uma comercial, voltada para o desenvolvimento de aplicações para fins comerciais, e uma gratuita, para o desenvolvimento de softwares livres.

Em março de 1996, Haavard e Eirik contrataram mais um desenvolvedor, e no mesmo ano, em setembro, foi lançada a versão 1.0 do Qt. Ainda em 1996, iniciou-se o projeto de desenvolvimento do *K Development Environment* (KDE, que será abordado em mais detalhes no tópico 2.3.2), liderado por Matthias Ettrich.

No início de 1997 foi lançado o Qt 1.2, e foi decidido que o KDE seria desenvolvido com o uso desse *framework*, o que consolidou o desenvolvimento GUI em C++ para Linux (BLANCHETTE; SUMMERFIELD, 2008).

Em 1998, a última atualização do Qt 1, a versão 1.40 foi desenvolvida, trazendo melhorias de performance.

Em junho de 1999, a versão 2.0 foi lançada, trazendo consigo um novo tipo de licença de código aberto, a *Q Public License* (QPL), que cumpria a definição de código aberto (*Open Source Definition*). Ainda em 1999, o *framework* Qt ganhou o prêmio LinuxWorld como melhor biblioteca/ferramenta.

De acordo com Blanchette e Summerfield (2008), no ano 2000 foi lançado o Qtopia Core (na época chamada de Qt/Embedded), criado para funcionar em sistemas Linux embarcados, e providenciava seu próprio sistema de janelas. No fim do mesmo ano foi lançada a primeira versão do Qtopia, para telefones móveis e *Personal Digital Assistant* (PDA’s). O Qtopia Core recebeu o prêmio de melhor solução para Linux embarcado por 2 anos seguidos, 2001 e 2002.

Em 2001 foi lançado o Qt 3.0, que estava disponível para Windows, Mac, Unix e Linux (*desktop* e embarcado). Nessa versão, foram implementadas 42 novas classes, e seu código ultrapassou 500.000 linhas.

* + - 1. QT 4

Em 2005, a versão 4.0 do Qt foi lançada, trazendo 5 novas tecnologias ao *framework*, sendo elas um conjunto de classes de *containers*, arquitetura *Model-View-Controller* (MVC), um *framework* de design gráfico 2D, um novo renderizador de texto Unicode e um novo tipo de janelas baseadas em ações (QT PROJECT, 2014l).

Ainda em 2005, a versão Qt 4.1 foi lançada, trazendo como novidades suporte à *Scalable Vector Graphics* (SVG) e um módulo back-end *Portable Document Format* (PDF) para o sistema de impressões do Qt.

Segundo Qt Project (2014l), em 2006 a versão 4.2 trouxe o suporte ao Windows Vista e suporte a *Cascading Style Sheets* (CSS) nativo, além do *framework* QgraphicsView para renderização eficiente de objetos 2D.

Em 2007, com a versão 4.3, veio uma melhoria no suporte ao Windows Vista, além de um suporte experimental para renderização de gráficos 3D através do OpenGL.

Em 2008, as maiores inovações na versão 4.4 do Qt foram a melhoria no suporte multimídia, o suporte a *eXtended Markup Language* (XML) e a compatibilidade com o Windows CE, sistema operacional da Microsoft para *tablets* e dispositivos móveis na época.

Na versão 4.5, lançada em 2009, novamente houveram melhorias nos motores gráficos e o Qt para Mac OS X teve várias partes reescritas para oferecer suporte à API Cocoa da Apple, permitindo que as aplicações desenvolvidas para esse sistema operacional pudessem funcionar em hardware Machintosh de 64-bits (QT PROJECT, 2014l).

Ainda em 2009, a versão 4.6 do *framework* foi lançada, trazendo suporte a gestos e *multi-touch* em dispositivos móveis, possibilitando a criação de interfaces intuitivas para os usuários. Ainda nessa versão foram introduzidos suporte ao sistema operacional *mobile* Symbian e suporte ao Windows 7.

No fim do ano de 2010, a versão 4.7 do Qt trouxe aos desenvolvedores o *Qt Meta-objects Language* (QML), uma linguagem de programação declarativa baseada em JavaScript. Com ela também foi introduzido o Qt Quick, módulo de desenvolvimento para aplicações com interfaces de usuário, que utiliza tanto o QML quanto o C++ como linguagens de desenvolvimento.

Por fim, em 2011 foi lançada a última atualização da versão 4 do Qt, a 4.8, onde basicamente foram trazidas melhorias de performance (QT PROJECT, 2014l).

* + - 1. QT 5

No final de 2012 foi lançada a mais recente versão do *framework*, o Qt 5. Com ele se tornou possível a criação de interfaces mais intuitivas para desenvolvimento em várias plataformas e suporte total à tecnologia de *touch screen*. Segundo Qt Project (2014m), essa nova versão também traz maior flexibilidade ao desenvolvedor, graças à melhoria na integração entre JavaScript, QML e C++.

Em julho de 2013, foi lançada a versão 5.1 do Qt, que trouxe suporte experimental para Android e iOS. No fim do mesmo ano, com o lançamento da versão 5.2 do *framework*, foi anunciado oficialmente o suporte para ambas plataformas.

Em maio de 2014 foi lançada a mais recente versão do Qt, a 5.3 que está focada em desempenho, estabilidade e usabilidade, além do suporte experimental para o Windows Runtime e a plataforma Windows Phone (QT PROJECT, 2014m).

Até o fim de 2014 pretende-se lançar a versão 5.4 do *framework*, e as principais novidades serão o suporte total à interface Windows RT, melhoria de motores gráficos e o suporte para que os componentes tenham a aparência nativa do Android.

O Qt tem aumentado cada vez mais sua popularidade, sendo a plataforma utilizada na criação de diversos softwares de renome, como Amazon Kindle, Google Earth, Guitar Pro, KDE, EA Origin, Oracle VirtualBox e o futuro Wireshark 2 (no momento em estágio *Preview*) (WIRESHARK, 2014a), como exemplificado na Figura 2:



Figura 2 – Programas que utilizam Qt

Fonte: Própria

Atualmente, o Qt é capaz de compilar aplicações para sistemas *desktop*, como Windows, Mac OS X e distribuições Linux, assim como algumas plataformas *mobile*, como Android, iOS, Windows CE e BlackBerry (QTPROJECT, 2014b). Segundo o Qt Project (2014a), está em processo de desenvolvimento o suporte completo para Windows Runtime (WinRT), permitindo a compilação para Windows Phone e utilização da interface MetroUI das versões Windows 8 e Windows 8.1.

* + 1. SINAIS E SLOTS (*SIGNALS AND SLOTS*)

O mecanismo de *signals* e *slots* foi criado por Eirik Chambe-Eng em 1992 (BLANCHETTE; SUMMERFIELD, 2008) e consiste em um dos princípios básicos do *framework* Qt, sendo incorporado por diversas *toolkits* desde então. De acordo com Qt Project (2014o), o conceito de tal funcionalidade é de permitir que dois objetos se comuniquem e façam um redirecionamento de uma ou mais funções instantaneamente, reduzindo e otimizando a quantidade de código-fonte criado em uma aplicação. Esse mecanismo só é possível em classes que derivam, direta ou indiretamente, de QObject.

A Figura 3 mostra o cabeçalho (*header*) de uma classe de exemplo com definições de *signal* e *slot*:



Figura 3 – Exemplo de classe com *signals* e *slots* implementados

Fonte: Ezust A; Ezust E; 2011

Com base em Thelin (2007) e Ezust e Ezust (2011), um *signal* se assemelha à um método de retorno vazio (*void*) sem implementação. Um *signal*, ao contrário dos métodos, não é executado, mas emitido pela instância do objeto onde ele se encontra. Ele faz parte da interface da classe. No exemplo acima, o *signal* textChanged, que passará como atributo o novo valor de m\_text, deverá ser emitido no método setText, e apenas quando o novo valor for diferente do atual (de modo a evitar *loops* infinitos na execução do código), como se pode ver na Figura 4:



Figura 4 – Implementação de emissão de *signal*

Fonte: Ezust A.; Ezust E., 2011

Ainda segundo Thelin (2007) e Ezust e Ezust (2011), um *slot* é um método sem retorno (*void*) que pode ser invocado quando um *signal* for emitido, além de ser usado como um método comum. É possível configurar diversos níveis de acesso à um *slot* (public, protected ou private), mas tal configuração apenas interfere em seu funcionamento como método. Quando usado como *slot*, este é sempre público.

Um *slot* pode chamar outro *signal*. Como exemplificado nas figuras acima, o método setText, quando recebe um texto diferente do atual, emite o *signal* textChanged, passando como parâmetro o texto que este recebeu.

As conexões entre objetos são realizadas através do método estático *connect*(), da classe QObject.Em Qt Project (2014p), é declarado que até a versão 4 do *framework* Qt, a sintaxe necessária para se criar tal conexão era a seguinte:

**connect(sender, SIGNAL(signal), receiver, SLOT(slot));**

Onde sender é o objeto emissor, signal é o método de sender que foi emitido, receiver é o objeto receptor, e slot é o método que receberá as informações passadas por signal.

Na versão 5 do Qt, foi introduzido uma nova sintaxe para realizar essa conexão:

**connect(sender, &Classe\_de\_Sender::signal, receiver, &Classe\_de\_Receiver::slot);**

Segundo Qt Project (2014p), o método antigo de se criar conexões continua sendo suportado na versão atual do *framework*, mas é passível de algumas falhas, sendo a principal delas o fato de a verificação sintática dos métodos connect() só ocorrer em tempo de execução (*runtime*), enquanto que, na sintaxe atual, esta ocorre durante a compilação, trazendo mais garantia de que a conexão foi implementada com sucesso.

A Figura 5 indica um exemplo entre ligações de vários objetos através de *signals* e *slots*, com um exemplo de código-fonte necessário para cada conexão:



Figura 5 – Exemplo do mecanismo de *signal* e *slot*

Fonte: Qt Project, 2014o

Um exemplo de utilização deste mecanismo é na utilização de *widgets*. Como exemplificado em Qt Project (2014o) e Molkentin (2006), em uma tela com um botão de “Fechar”, é desejável que a tela feche quando o usuário clicar nesse botão. Nesse caso, uma solução seria realizar uma conexão entre a função de clique do botão e o método quit() da tela, como exemplificado na figura abaixo (Figura 6):



Figura 6 – Exemplo de código-fonte com conexão entre objetos utilizando *signals* e *slots*

Fonte: Molkentin, 2006.

Implementando o código-fonte presente na imagem acima, é possível fazer a seguinte análise:

* button é o objeto que será responsável por emitir o comando de clique do botão.
* clicked() é uma função de button, emitida quando o botão foi clicado.
* a é o objeto responsável por receber a ação realizada por clicked().
* quit() é uma função de a. Ela representa o ato de encerrar a aplicação, e será acionada sempre que a função clicked() de button for executada.

Segundo Ezust e Ezust (2011), nas conexões entre objetos também é possível conectar o *signal* de um objeto ao *signal* de outro objeto, de modo que este segundo *signal* possa acionar seus *slots* associados. Outra característica das conexões que é descrita em Qt Project (2014o), Thelin (2007) e Molkentin (2006) é relacionada à quantidade de argumentos dos *signals* e *slots*: O *slot* não pode possuir, em sua assinatura de método, argumentos a mais ou em ordem diferente dos que foram passados pelo *signal*. Caso tal comportamento ocorra, é gerado um erro na execução do programa e a conexão não é estabelecida. Em contrapartida, o *slot* ignorará quaisquer argumentos a mais que o *signal* associado transmitir.

* 1. EXEMPLOS DE APLICAÇÕES QUE UTILIZAM QT

Neste tópico, são apresentadas algumas aplicações onde foi utilizado o *framework* Qt para seu desenvolvimento, como por exemplo o Wireshark, uma ferramenta de captura de pacotes de rede, e o principal ambiente de trabalho para Linux, a KDE.

* + 1. WIRESHARK

O Wireshark é um analisador de protocolos de rede, sendo considerado o principal *software* do ramo. Ele foi inicialmente criado em 1997 por Gerald Combs, formado em ciência da computação pela Universidade do Missouri-Kansas City, sob o nome de Ethereal, nome que foi mudado para Wireshark em 2006 (WIRESHARK, 2014b).

De acordo com o Wireshark (2014b), o desenvolvimento da ferramenta começou quando Combs estava precisando rastrear um problema de rede na empresa onde trabalhava, e não havia encontrado uma ferramenta que lhe servisse, então decidiu criar seu próprio *software*.

Seu primeiro lançamento foi na versão 0.2.0 em julho de 1998. Em questão de meses, começaram a surgir mais e mais pessoas dispostas a contribuir com o projeto, seja com contribuições realizadas no código-fonte ou inserindo novos decodificadores de protocolos (também chamados de *dissectors* pelo Wireshark). Segundo Wireshark (2014c), atualmente, a lista de colaboradores já passa de 800 pessoas.

Até o presente momento, o Wireshark é desenvolvido utilizando a ferramenta de desenvolvimento multiplataforma GTK+ (*GIMP Toolkit*), mas, a partir da versão 1.11, foi iniciada a migração do código-fonte para o *framework* Qt, de modo a unificar a interface gráfica entre as plataformas (Gerald Combs alega que a versão do Wireshark para o Mac OS X não “aparenta ou age de forma alguma como uma aplicação para Mac OS X”) (WIRESHARK, 2014d).

Desde então, ao instalar a versão mais atual do Wireshark, são instaladas as duas versões (Wireshark original e Wireshark 2 *Preview*), de modo que os usuários possam se acostumar à nova interface. A Figura 7 indica o estado atual da nova interface do Wireshark em funcionamento:



Figura 7 – Wireshark 2 Preview em funcionamento

Fonte: Própria

* + 1. KDE

O KDE é um ambiente de trabalho criado inicialmente para os sistemas operacionais baseados no Unix, e que atualmente faz parte da comunidade KDE, uma comunidade voltada para o desenvolvimento de aplicações de código livre e *Open Source* (KDE, 2014c). A comunidade KDE é estruturada conforme a Figura 8:



Figura 8 – A comunidade KDE

Fonte: KDE, 2014c

Segundo KDE (2014a), o KDE foi projetado em 1996 por Matthias Ettrich, que se encontrava insatisfeito com os ambientes de trabalho existentes na época para os sistemas Unix. Ele alegava que era necessária a existência de uma GUI que oferecesse uma aparência e sensação comum para todas as aplicações. Portanto, iniciou o projeto KDE em busca de apoio para que tal ambiente fosse criado.

Desde o princípio do projeto, Ettrich especificou que seria utilizado o Qt para o desenvolvimento de sua ferramenta, devido à sua facilidade na utilização de bibliotecas C++ e portabilidade. Sua preferência pelo Qt alavancou ainda mais a popularidade deste *framework*, além de firmar o Qt como a principal ferramenta para o desenvolvimento de aplicações GUI em C++ (BLANCHETTE; SUMMERFIELD, 2008).

Devido ao fato de utilizar o Qt como base, o KDE permite que as aplicações criadas nele possam ser compiladas para diversos sistemas operacionais. Existem versões da ferramenta (e de suas aplicações) para sistemas Unix, Mac OS X e Windows. (KDE, 2014c).

Segundo KDE (2014b), o ambiente de trabalho do KDE (e seu principal produto) se chama Plasma, e é composto pela interface gráfica exemplificada na Figura 9:



Figura 9 – Interface do Plasma

Fonte: KDE, 2014b

Um dos conceitos principais do Plasma é o dos *widgets*. Segundo KDE (2014b), *widgets* são unidades visuais individuais e interdependentes que podem ser posicionadas no ambiente de trabalho. Podem ser adicionados, removidos, redimensionados e interagidos de diversas formas, e podem ter as mais variadas utilidades, como: previsão do tempo, calculadora, compartilhamento de arquivos, entre outros. O Plasma suporta a inserção e utilização de *widgets* feitos especificamente para ele (chamados Plasmóides), assim como provenientes de terceiros, como Google Gadgets e Dashboard, do Mac OS X.

Um ambiente de trabalho padrão do Plasma possui os seguintes componentes:

* Painel: contém *widgets* como o relógio, a área de notificação e a barra de tarefas, além de permitir o posicionamento de *widgets* adicionais nas bordas da tela.
* Caixa de ferramentas do Plasma: localizado no topo direito da tela e à direita do painel, permite alterar as configurações da ferramenta.
* Exibição de Pasta: uma janela que possui visão configurável para qualquer pasta do sistema (o padrão é a área de trabalho do usuário), que é onde se pode manipular os arquivos da pasta em questão. É permitido possuir mais de uma exibição de pasta na área de trabalho, assim como posicioná-las nos paineis.
* Área de trabalho: a tela como um todo, onde os *widgets* e ícones estão presentes.

Segundo KDE (2014d), em 1998 foi fundada a *KDE Free Qt Foundation*, uma organização criada pela representação legal do KDE, KDE e. V., e a Trolltech (criadora do *framework* Qt). Esta fundação tem o propósito de garantir que a ferramenta Qt continue disponível pelas licenças LGPL e GPL, garantindo assim sua utilização para a criação de softwares livres, mais especificamente a ferramenta KDE. O acordo firmado prevê que, caso a detentora dos direitos do Qt cesse tal disponibilidade, a fundação tem todo o direito de liberar a utilização do Qt sob uma licença *open source*.

* 1. QT CREATOR

Qt Creator (Figura 10) é um ambiente de desenvolvimento integrado (IDE) que provê ao usuário ferramentas para modelar e desenvolver aplicações com o *framework* de aplicação Qt (QT PROJECT, 2014c).



Figura 10 – Tela inicial do Qt Creator

Fonte: Própria

Segundo o Qt Project (2014c), o Qt Creator provê dois editores visuais integrados, Qt Quick Designer e Qt Designer, cada um responsável por gerenciar os dois módulos de interface gráfica do Qt: Qt Quick e Qt Widgets

O Qt Creator possui duas versões, uma delas totalmente grátis, que permite a criação de aplicações para plataformas *desktop* (Windows, Linux e Mac) e *mobile* (Android, iOS e mais recentemente Windows Phone 8). A outra versão, denominada Qt Creator Enterprise, oferece uma gama ainda maior de plataformas, incluindo sistemas embarcados, como computadores de bordo de carros **(procurar foto)**, painéis digitais e até mesmo possibilita a criação de aplicações de missão crítica, que funcionam em sistemas de tempo real, onde o tempo de resposta deve ser constante e pré-definido. (QT DIGIA, 2014a).

* + 1. QT QUICK

O Qt Quick (*Qt User Interface Creation Kit*) é um dos submódulos do *framework* Qt. Ele teve seu surgimento na versão 4.7 do *framework* e foi criado pensando em otimizar a criação de interfaces de usuário para plataformas mais responsivas, principalmente os sistemas operacionais *mobile* (QT PROJECT, 2014d).

Uma das particularidades do Qt Quick é relacionada ao modo de programação utilizando tal modo. O código-fonte de uma aplicação do Qt Quick não precisa se basear apenas no C++, pois este módulo introduziu também a linguagem QML, que será tratada com mais detalhes no próximo tópico.

* + - 1. QML

QML é uma linguagem declarativa que faz parte do *framework* Qt. QML é utilizada no desenvolvimento de aplicativos *cross-platform*[[3]](#footnote-3) e busca facilitar o projeto e a implementação de interfaces de usuário (UIs) para dispositivos móveis através da rapidez na codificação e na prototipagem. (ROSA et al., 2011). Ela possibilita a criação de interfaces fluidas e animadas, além de integração com bibliotecas em C++.

Para Rosa et al. (2011) e Qt Project (2014e), o estilo de programação da linguagem QML é baseado nas linguagens CSS e JavaScript, tornando-se de aprendizado rápido e fácil para programadores C, Qt/C++, Java e principalmente desenvolvedores web. O QML permite o uso de funções em JavaScript para sua lógica, e propriedades entrelaçadas (quando o valor de uma propriedade é relativo ao valor de outra propriedade), como ilustrado na Figura 11:



Figura 11 – Exemplo de propriedades atreladas à outras (*property bindings*)

Fonte: Rosa et al., 2011

Os objetos QML são especificados por meio de seus elementos e cada um deles 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., como exemplificado na Figura 12. 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. (ROSA et al., 2011)



Figura 12 – Exemplo de código em QML para criação de um retângulo

Fonte: Rosa et al., 2011

Uma aplicação QML é executada através da máquina de execução QML, também chamada de QML runtime. 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 Viewer. (ROSA et al., 2011)

A Figura 13 ilustra alguns dos tipos de dados utilizados em uma aplicação QML:



Figura 13 – Alguns tipos utilizados no sistema de tipagem da linguagem QML.

Fonte: Rosa et al., 2011

A linguagem QML e sua *engine* de infraestrutura são disponibilizadas através do módulo Qt QML, um *framework* para o desenvolvimento de aplicações e bibliotecas utilizando a linguagem QML, além de prover uma API para estender a linguagem com tipos customizados e integrar um código em QML com JavaScript e C++ (QT PROJECT, 2014h).

Entretanto, Qt Project (2014h) também cita que, enquanto o módulo Qt QML provê a linguagem e a infraestrutura para aplicações em QML, o módulo Qt Quick oferece vários componentes visuais, suporte à arquitetura Modelo-View (*Model-View Architecture*), *framework* de animação, e muitas outras funcionalidades para gerar interfaces com usuário.

* + 1. QT WIDGETS

Segundo Qt Project (2014f)*, widgets* são os componentes fundamentais para a criação de UIs no Qt. Eles provêm *layout* e comportamento no estilo padrão dos componentes de aplicações *desktop*, e podem servir para exibir dados de saída, receber dados de entrada do usuário, ou como *containers* para outros *widgets* se aninharem. A Figura 14 mostra um exemplo de agrupamento de *widgets* em uma janela, e suas respectivas classes:



Figura 14 – Exemplo de tela com *widgets* aninhados

Fonte: Qt Project, 2014f

O módulo Qt Widgets está presente no Qt desde as primeiras versões, sendo o módulo mais maduro para a criação de interfaces gráficas. Segundo Qt Project (2014j), seu intuito, ao contrário do módulo Qt Quick, é de oferecer interfaces de usuário com a aparência nativa da plataforma, por isso ele é mais recomendado para ser utilizado em aplicações voltadas para as plataformas *desktop*, como o Windows, Mac OS X e as distribuições Linux. Isso se deve ao uso de Estilos (*Styles*) na interface de usuário. A Figura 15 exemplifica alguns estilos de interface de usuário do Qt (podendo ser configurados através da classe QStyle). Vale lembrar que os estilos para Windows e Mac OS X só são disponíveis em suas respectivas plataformas (BLANCHETTE; SUMMERFIELD, 2008; QT PROJECT, 2014f).



Figura 15 – Exemplo de Estilos dos *Widgets*.

Fonte: Blanchette; Summerfield, 2008

Outra funcionalidade importante do módulo Qt Widgets é o conceito de *Layouts* e Gerenciadores de *Layouts* (*Layout Managers*). Para Blanchette e Summerfield (2008), o gerenciador de *layouts* é “um objeto que define o tamanho e a posição dos *widgets* que estão sob sua responsabilidade”. O Qt possui diversos gerenciadores de *layout* (vide Figura 16), dentre eles podem-se citar:

* QHBoxLayout: Alinha os *widgets* horizontalmente
* QVBoxLayout: Alinha os *widgets* verticalmente
* QGridLayout: Alinha os *widgets* em células dentro de um *grid*. É possível configurar de modo que um *widget* possa ocupar mais de uma célula.
* QFormLayout: Alinha os *widgets* no formato padrão de dois objetos por linha, visando a utilização do padrão “descrição – campo”.



Figura 16 – Exemplos de gerenciadores de *layouts* do módulo Qt Widgets

Fonte: Qt Project, 2014n

Os gerenciadores de *layout* ajustam automaticamente os objetos-filho assinalando valores de posicionamento e dimensões, de modo que a interface como um todo mantenha sua proporção e usabilidade (BLANCHETTE;SUMMERFIELD, 2008; QT PROJECT, 2014n).

1. ALTERNATIVAS AO QT
   1. INTRODUÇÃO

Nesse capítulo serão apresentados alguns exemplos de ferramentas e linguagens de programação que possuem o mesmo propósito do Qt, que é o de facilitar o desenvolvimento multiplataforma.

* 1. XAMARIN

Xamarin é uma plataforma de desenvolvimento multiplataforma voltada principalmente para dispositivos móveis, embora também ofereça suporte para alguns sistemas operacionais desktop. Ele foi desenvolvido tendo por base a plataforma Mono, que também é voltada para o para a criação de softwares multiplataforma. Ambos, Xamarin e Mono utilizam a linguagem C# (lê-se C *Sharp*) para o desenvolvimento, embora o Mono apresente suporte para outras linguagens também (MONO, 2014) e, enquanto o Mono baseia-se no próprio .NET *framework*, o Xamarin utiliza bibliotecas e APIs próprias. Mais informações sobre a plataforma Mono se encontram no tópico 3.3.

A plataforma Xamarin foi criada e é suportada pela empresa que leva o mesmo nome, sendo que sua sede principal está localizada em São Francisco, Califórnia. A empresa possui, segundo dados próprios (XAMARIN, 2014a), 170 funcionários que se encontram em 14 países diferentes, aproximadamente 15000 clientes, em 120 países e mais de 700000 desenvolvedores utilizando sua plataforma de desenvolvimento. A Figura 17 apresenta o logo da empresa.



Figura 17 – Logo da empresa Xamarin

Fonte: Xamarin, 2014a

As plataformas suportadas pela Xamarin são: iOS, Android, Windows Phone, Mac e Windows (*desktop*). Segundo dados da própria empresa (XAMARIN, 2014b), utilizando-se a mesma linguagem de programação, no caso, o C#, as mesmas APIs e as mesmas estruturas de dados, é possível reaproveitar aproximadamente 75% do código-fonte ao portar-se o *software* desenvolvido de uma plataforma para outra, sendo necessário apenas a reconstrução das interfaces de usuário de cada plataforma desejada. Com a utilização da API Xamarin.Forms (vide tópico 3.2.1), esse reaproveitamento se aproxima dos 100%.

* + 1. XAMARIN.FORMS

A Xamarin.Forms é uma API feita para que os desenvolvedores não precisem se preocupar com a criação de diversas interfaces, uma para cada sistema operacional em que se deseje que o aplicativo rode. A Xamarin.Forms é voltada somente para o desenvolvimento de aplicativos para dispositivos móveis, sendo as plataformas suportadas: Android, iOS e Windows Phone. A API utiliza um conceito onde a interface é desenvolvida utilizando-se *layouts* e controles comuns (como botões, *labels*, listas, entre outros) que posteriormente são combinados com códigos da própria API, formando componentes nativos para as plataformas.

A Figura 18 ilustra um trecho de código para a criação de uma interface básica de *login*, e a Figura 19 mostra como essa tela é apresentada por cada um dos sistemas operacionais.



Figura 18 – Código para a criação de uma tela de *login* comum

Fonte: Xamarin, 2014c



Figura 19 – Código mostrado na Figura 18 rodando, respectivamente, em iOS, Android e Windows Phone

Fonte: Xamarin, 2014c

Existem ainda APIs voltadas completamente para o Android (Xamarin.Android) e para o iOS (Xamarin.iOS), e, segundo a empresa (XAMARIN, 2014c), códigos desenvolvidos utilizando-se a Xamarin.Forms podem ser combinados com códigos das APIs específicas para cada sistema operacional, potencializando o desenvolvimento dos aplicativos.

* + 1. IDEs

A plataforma Xamarin pode ser utilizada em duas IDEs: uma delas é o Visual Studio, da Microsoft, e a outra é uma IDE própria, chamada Xamarin Studio.

O Xamarin para Visual Studio permite a criação de aplicativos com interface gráfica nativa para as plataformas Apple iOS, Google Android e para o próprio Windows. Utilizando a tecnologia denominada IntelliSense que o Visual Studio possui (que mostra sugestões para que o código se auto-complete, trazendo descrições como parâmetros de métodos, sua descrição, sobrecargas, entre outras informações), é possível tornar o desenvolvimento mais rápido e descobrir funcionalidades das APIs para Android e iOS de maneira mais simples.

Para que se possa utilizar o Xamarin no Visual Studio, é necessário fazer o download e instalação do *plugin* do Xamarin para a IDE, que pode ser encontrado no próprio *site* da empresa. Existem alguns requisitos que devem ser atendidos para que o Xamarin possa ser utilizado, sendo eles a necessidade de utilizar Windows 7 ou alguma versão posterior, Visual Studio 2010 *Professional* ou posterior (versões *Express* do Visual Studio não são compatíveis, pois não aceitam *plugins*) e o *plugin* do Xamarin. (Xamarin, 2014d)

No entanto, não é possível compilar um aplicativo para iOS sem o compilador da Apple, assim como não é possível fazer o *deploy* (implantação na plataforma de destino) de uma aplicação sem os certificados e assinaturas digitais da mesma empresa. O Windows não possui suporte à tais atividades, portanto o Visual Studio precisa que a máquina Windows onde está instalado possua conexão via rede à alguma máquina Mac que possua o sistema operacional Mac OS X na versão 10.8, ou mais recente, que também possua o Xamarin instalado, para que essa segunda máquina possa executar os procedimentos necessários, visto que ela possui os recursos adicionais necessários.

O Xamarin Studio oferece suporte ao desenvolvimento de aplicações para iOS, Android e Mac, e está disponível tanto para Windows quanto para Mac OS X. Na versão para Windows porém, só é possível o desenvolvimento de aplicações para Android e para o próprio Windows. Para desenvolver-se algo para iOS ou Mac OS é necessário utilizar o Xamarin Studio instalado em uma máquina que possua o sistema operacional da Apple instalado e utilizando a API Xamarin.iOS. Como citado anteriormente, o *plugin* do Xamarin para a IDE Visual Studio da Microsoft fornece suporte ao desenvolvimento de tais aplicações, desde que possa se conectar à uma máquina com o Mac OS X instalado via rede.

O Xamarin Studio apresenta funcionalidades como código que se auto-completa durante a digitação, ferramentas para a publicação de aplicativos diretamente na App Store da Apple ou na Google Play Store, além de um *debugger* completo. O Xamarin Studio ainda possui módulos de *design* de interfaces para Android e iOS, e através das APIs Xamarin.Android e Xamarin.iOS, além da Xamarin.Forms, que permitem a criação de aplicações com interfaces e desempenho de aplicações nativas. (Xamarin, 2014e)

A Figura 20 representa a criação de uma interface gráfica para iOS no Visual Studio da Microsoft, enquanto a Figura 21 representa a interface do Xamarin Studio.



Figura 20 – Criação de interface para iOS utilizando-se o Xamarin para Visual Studio

Fonte: Xamarin, 2014b



Figura 21 – Criação de interface para iOS utilizando-se o Xamarin Studio

Fonte:Xamarin, 2014e

No próprio *site* da Xamarin é possível fazer o *download* dos pacotes básicos de desenvolvimento, tanto para Mac OS X quanto para Windows, sendo que em ambos estão inclusos o Xamarin Studio e as APIs Xamarin.iOS e Xamarin.Android. Para Windows o pacote é completado com o *plugin* de integração do Xamarin para Visual Studio, e para o Mac OS X o item final é a API Xamarin.Mac, que permite o desenvolvimento de aplicações para a plataforma.

* + 1. CLIENTES DA XAMARIN

A Xamarin, como já mencionado (XAMARIN, 2014a), possui aproximadamente 15000 clientes ao redor do mundo, e esse tópico apresenta exemplos de dois dos principais clientes: Rdio e Kimberly-Clark.

Rdio é um serviço de *streaming* de músicas que possui um grande catálogo, de aproximadamente 25 milhões de títulos. Quando foi lançado, ele possuía 3 códigos-fonte diferentes, um para iOS, um para Android e outro para Windows Phone, o que dificultava atualizações e manutenções nos códigos. Em dezembro de 2012 eles migraram o aplicativo para uma nova versão desenvolvida com Xamarin.

Após a atualização, houve um ganho de desempenho, pois os aplicativos são nativos de cada plataforma, e houve uma facilitação nas questões de atualizações e manutenções, pois mais de 50000 linhas de código eram comuns para todas as plataformas (XAMARIN, 2014f). A Figura 22 representa o Rdio em um iPad rodando iOS e um *smartphone* rodando Android.



Figura 22 – Software Rdio rodando em iOS (tablet) e Android (smartphone)

Fonte: Xamarin, 2014f

A Kimberly-Clark, por sua vez, desejava aumentar as vendas de seus produtos, voltados para higiene e bem-estar, a partir do uso de iPads para criar propostas personalizadas para os clientes. A empresa já havia tentado a utilização de um aplicativo desenvolvido em HTML5 para isso, porém sem sucesso.

A Kimberly Clark então trocou para o Xamarin, que permitiu que eles pudessem reaproveitar grande parte do código já existente em C# além dos serviços *web*, o que proporcionou muitas horas economizadas em desenvolvimento.

Com o novo aplicativo, desenvolvido com Xamarin, a Kimberly-Clark conseguiu um aumento significativo em sua receita e uma redução no tempo de vendas, o que proporcionou uma maior satisfação para os clientes. Segue o relato de Kim MacDougall, gerente sênior de desenvolvimento de capacidades **(Ver melhor tradução para esse termo)** da Kimberly-Clark:

Os resultados obtidos a partir de nosso novo aplicativo de vendas de campo são fenomenais – nossa equipe de vendas ama o aplicativo e são capazes de conquistar os clientes e fechar negócios de maneira mais eficiente. A chave para o sucesso do aplicativo é a bela e rápida experiência do usuário possibilitada pelo Xamarin. (XAMARIN, 2014f).

A Figura 23 mostra a interface do aplicativo desenvolvido com Xamarin que a Kimberly-Clark utiliza na área de vendas de sua empresa.



Figura 23 – Aplicativo da empresa Kimberly-Clark desenvolvido com utilização do Xamarin

Fonte: Xamarin, 2014f

* + 1. XAMARIN TEST CLOUD

O Xamarin Test Cloud é um serviço oferecido pela empresa baseado no modelo de computação em nuvem para que os desenvolvedores possam criar rotinas de testes automatizados, principalmente para realizar os testes de aceitação de interface gráfica, que são conhecidos por serem custosos (Xamarin, 2014g). Esse teste consiste em instalar o aplicativo desenvolvido em vários aparelhos diferentes que rodem a mesma plataforma, e analisar o comportamento do *software* em cada um dos equipamentos. O teste de aceitação é importante para que se possam corrigir erros que eventualmente apareçam para um aparelho em específico, antes da liberação do aplicativo para uso público, maximizando a satisfação do usuário, independentemente de qual aparelho ele possua.

O serviço disponibiliza aos desenvolvedores centenas de plataformas diferentes para que os aplicativos desenvolvidos possam ser testados, tanto para iOS quanto para Android, desde aparelhos mais antigos, com versões obsoletas dos sistemas operacionais, até os aparelhos mais recentes com versões atuais das plataformas.

Os testes automatizados podem ser desenvolvidos utilizando-se dois *frameworks* oferecidos pelo Xamarin: o Xamarin.UITest, que permite a criação de rotinas de teste em C# e é mais voltado para desenvolvedores experientes na criação de testes automatizados, e o Calabash, que permite a escrita das rotinas de teste na linguagem Ruby utilizando o Cucumber, que é um *software* voltado para testes automatizados.

Após escritos os testes, os desenvolvedores devem submeter seu aplicativo aos servidores da Xamarin, onde ele será submetido aos testes nos aparelhos desejados pelo desenvolvedor, e depois de feitos os testes, o serviço gera relatórios com resultados obtidos nos testes, *screenshots* e métricas de desempenho, para que o desenvolvedor possa analisá-los e fazer as correções que achar necessárias (Xamarin, 2014h).

A Figura 24 representa a interface do serviço Test-Cloud, para testes automatizados em nuvem.



Figura 24 – Interface do Xamarin Test Cloud

Fonte: Xamarin, 2014h

* 1. MONO

Mono é uma plataforma de desenvolvimento de código aberto para softwares multiplataforma baseada no .NET *framework* da Microsoft. A Mono é suportada pela empresa Xamarin, já mencionada no tópico 3.2 dessa monografia, e um de seus diferenciais é que oferece suporte a mais de uma linguagem diferente, sendo alguns dos exemplos que os desenvolvedores podem escolher: C#, F#, *Visual Basic*, Java, JavaScript, Object Pascal, entre outras. (MONO, 2014a) A plataforma Mono oferece tanto ferramentas de desenvolvimento quanto a infraestrutura necessária para rodar aplicações clientes e servidores .NET.

A Figura 25 apresenta o logotipo da Mono:



Figura 25 – Logotipo da Mono

Fonte: Mono, 2014a

* + 1. COMPONENTES DA MONO

A plataforma Mono é composta por alguns componentes, sendo eles: um compilador para C#, uma máquina virtual própria para rodar os códigos gerados, uma biblioteca de classes básicas e outra biblioteca que oferece classes com funcionalidades adicionais.

O compilador C# oferecido pela plataforma oferece suporte completo para todas as versões da linguagem, desde a versão 1.0 até a versão 5.0 a mais atual no momento, embora já esteja sendo distribuída uma prévia da versão 6.0 da linguagem, com a versão 3.8 da Mono. Segundo dados próprios (MONO, 2014b), tal compilador é capaz de compilar a si mesmo, além de outros códigos em C#. Esse compilador também costuma ser utilizado para compilar a própria Mono, que possui cerca de 4 milhões de linhas de código, além de ser rápido, capaz de compilar aproximadamente 18000 linhas de código por segundo em uma máquina comum.

A máquina virtual oferecida pela plataforma, a Mono Runtime, possui dois tipos de compiladores, um deles utiliza a tecnologia JIT (*Just-in-Time*), onde o código é compilado durante o tempo de execução do programa, ou seja, enquanto o programa roda ele vai sendo compilado, e o outro utiliza a tecnologia AOT (*Ahead-of-Time*), que faz uma pré-compilação do código assembly, o que faz com que o que minimiza o tempo de compilação JIT, reduz o uso de memória durante a execução do código e ainda proporciona um maior compartilhamento de código entre múltiplas aplicações Mono rodando. (MONO, 2014c).

Além disso, a Mono Runtime ainda oferece alguns serviços, como a execução do código, um sistema de coletor de lixo, que remove dados desnecessários da memória do computador quando deixam de ser usados pelo software, geração de código, conforme os compiladores apresentados no parágrafo anterior, tratamento de exceções, interface com o sistema operacional e gerenciamento de *threads*. A Mono Runtime ainda conta com a vantagem de poder funcionar em sistemas embarcados. (MONO, 2014d).

Ainda como componentes essenciais da Mono se encontram uma biblioteca de classes básicas, compatíveis com as classes do .NET *framework*, que é fornecida pela Microsoft e ajuda na criação de uma base sólida para o desenvolvimento, e uma biblioteca de classes da própria Mono, que oferece algumas funcionalidades adicionais, úteis para desenvolvimento para a plataforma Linux.

* + 1. SISTEMAS OPERACIONAIS SUPORTADOS

*Softwares* desenvolvidos com a Mono podem funcionar tanto em sistemas operacionais 32-bits quanto em 64-bits, assim como em uma gama de arquiteturas diferentes.

Os sistemas operacionais suportados são: Linux, Mac OS X, iOS, Solaris, BSD, Windows, Nintendo Wii e Playstation 3, sendo que para o último, o suporte é prestado para o Playstation 3 rodando Linux, uma versão da Mono compatível para o sistema operacional nativo do Playstation 3 está em desenvolvimento, porém sem data determinada para lançamento. (MONO, 2014e).

A Figura 26 apresenta uma tabela retirada do site da própria Mono, que mostra as arquiteturas suportadas e os respectivos sistemas operacionais.



Figura 26 – Plataformas e arquiteturas suportadas

Fonte: Mono, 2014e

No entanto, as arquiteturas Alpha, MIPS, ARM e HPPA são suportadas pela comunidade de desenvolvedores, estando sujeitas a não estarem tão completas quanto as demais arquiteturas.

Uma das principais vantagens da Mono é que, por utilizar o conceito de rodar sobre uma máquina virtual, no caso a Mono Runtime (vide tópico 3.3.1), é possível utilizar compilar códigos para funcionar em sistemas operacionais embarcados.

* + 1. BENEFÍCIOS

Os principais benefícios que a plataforma oferece, citados pela própria Mono (MONO, 2014a), são: a popularidade, utilização de linguagens de programação de alto nível, a sólida biblioteca de classes básicas, a possibilidade de desenvolvimento multiplataforma e a CLR (*Common Language Runtime*).

A popularidade se deve ao fato de que a plataforma se baseia no .NET *framework*, que é utilizada vastamente por desenvolvedores, inclusive utilizando a linguagem de programação C#, sendo possível encontrar-se materiais diversos como livros, guias e exemplos na internet.

A utilização da Mono Runtime permite ao desenvolvedor, independentemente da linguagem escolhida, desde que suportada pelo Mono, focar na criação da aplicação em si, pois ela oferece alguns componentes que permitem ao desenvolvedor abstrair de questões relacionadas à infraestrutura do código, como sistema de coletor de lixo, gerenciamento de memória, gerenciamento de threads entre outros.

A biblioteca de classes básicas oferecida fornecem milhares de classes já construídas, possibilitando um ganho de velocidade ao processo de desenvolvimento de um *software*, pois o programador economiza o tempo que levaria para escrever tais classes, utilizando uma pronta.

Como já citado no tópico 3.3.2, o Mono foi projetado para oferecer suporte a desenvolvimento multiplataforma, permitindo que o desenvolvedor também possa de certa forma se abstrair dessa questão, pois mesmo que sejam necessárias adaptações no código-fonte para fazê-lo rodar em outra plataforma ou arquitetura, não é necessário reescrevê-lo totalmente.

A CLR (*Common Language Runtime*) permite ao desenvolvedor escolher a linguagem que preferir, e permite que ela interopere com códigos escritos em outras linguagens, desde que ambas possuam o suporte à CLR.

* + 1. IDE

A IDE MonoDevelop é um ambiente de desenvolvimento criado para a programação utilizando a plataforma Mono, e foi feito para utilizar exclusivamente linguagens provindas do .NET, sendo a principal o C#.

Segundo o site oficial da própria IDE (MONO, 2014f), ela permite que os desenvolvedores criem rapidamente *softwares* tanto para *desktop* quanto para web, com ASP.NET, nas plataformas Windows, Linux e Mac OS X, além de facilitar a portabilidade de projetos criados no Microsoft Visual Studio para que rodem em Linux e Mac OS X mantendo o mesmo código base para todas as plataformas.

As principais funcionalidades apresentadas nessa IDE são o suporte a plataformas variadas, sendo elas Windows, Linux e Mac OS X, uma ferramenta para que o código da versão 4.0 da linguagem C# se auto-complete, layouts de janelas totalmente customizáveis, suporte a várias linguagens, entre elas C#, Visual Basic.NET, C e C++, um debugger para realizar o debug tanto de aplicações Mono quanto aplicações nativas, o módulo GTK# *Virtual Designer* para o desenvolvimento de interfaces, e a possibilidade de criação de projetos ASP.NET.

A Figura 27 apresenta a interface da página principal da MonoDevelop, enquanto a Figura 28 apresenta a tela de criação de interfaces.



Figura 27 – Tela principal do MonoDevelop

Fonte: Mono, 2014f

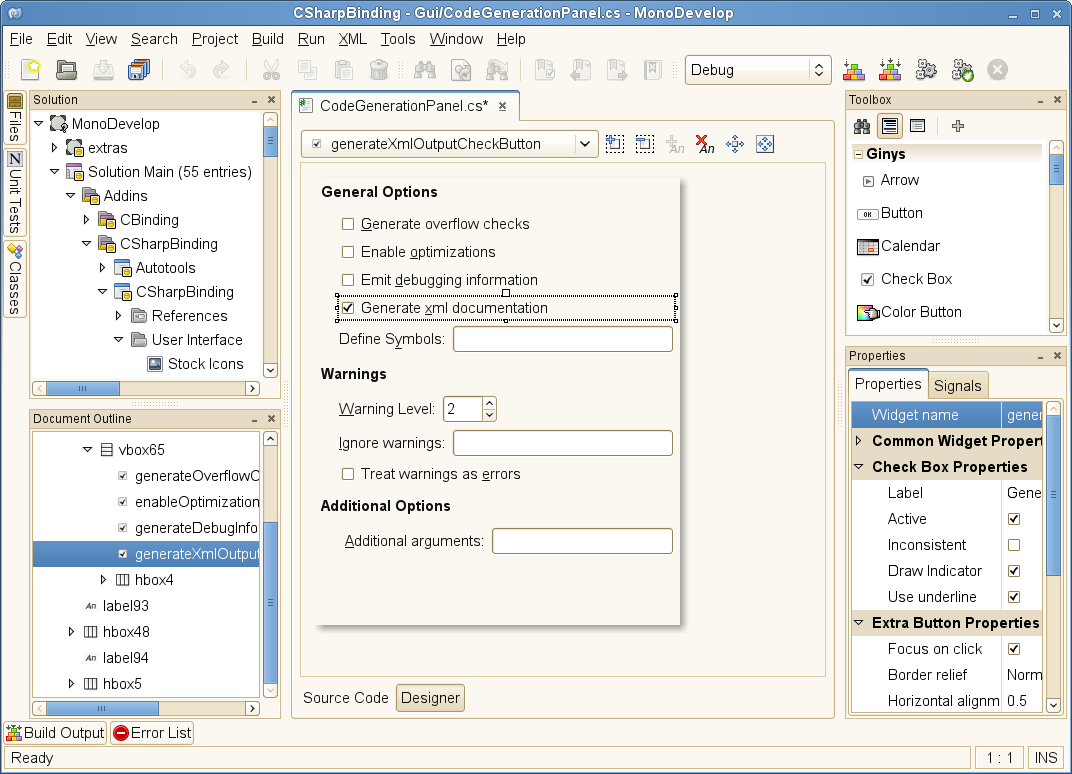


Figura 28 – Tela de criação de interfaces da IDE MonoDevelop

Fonte: Mono, 2014f

* 1. JAVA

Java é uma linguagem de programação orientada a objetos, desenvolvida pela empresa Sun Microsystems, adquirida posteriormente pela Oracle, em meados da década de 1990, que foi desenvolvida visando solucionar alguns dos principais problemas dos desenvolvedores dessa época, sendo um deles a necessidade da reescrita de grande parte dos códigos-fonte de *softwares* criados para rodar em mais de um sistema operacional distinto. Para tornar isso possível, foi utilizado o conceito de máquina virtual, que será explicado ainda nesse capítulo.

Segundo Deitel e Deitel (2012) e como consta no próprio site do Java (2014), em uma conferência no ano de 2010, a Oracle fez um anúncio em que dizia que 97% dos computadores corporativos do mundo utilizavam Java, além de mais de 3 bilhões de dispositivos portáteis e 80 milhões de aparelhos de televisão.

A Figura 29 apresenta o logo do Java:



Figura 29 – Logotipo do Java

Fonte: Java, 2014

* + 1. HISTÓRIA DO JAVA

O Java foi desenvolvido pela empresa Sun Microsystems, como resultado de um projeto interno, liderado por James Gosling, que é considerado o “pai” do Java. Segundo o curso Caelum (2014a), em 1992 um time liderado por Gosling, chamado *Green Team*, teve a ideia de desenvolver um interpretador para aparelhos eletrônicos como televisão, vídeo cassete e aparelhos de televisão a cabo, para facilitar a reescrita de código. A ideia porém não vingou e embora tenham tentado fechar contrato com grandes fabricantes como a Panasonic, não obtiveram sucesso devido à questões de custos.

Apesar disso, com o aumento da popularidade da internet, a Sun viu uma oportunidade, percebendo que poderiam utilizar a tecnologia Java para rodar pequenas aplicações (*applets*) nos *browsers* disponíveis. Isso apresentava alguma semelhança com o projeto original, pois na internet havia uma grande variedade de sistemas operacionais e browsers, fazendo com que a vantagem de se programar com uma só linguagem sem depender da plataforma fosse exposta.

No ano de 1995 o Java em sua versão 1.0 foi lançado oficialmente pela Sun, com o objetivo de tornar os *browsers* em algo que pudesse realizar operações mais avançadas, não somente renderizar e exibir páginas html.(Caelum, 2014a).

No ano de 2009 a Oracle, representada na Figura 30, adquiriu a Sun Microsystems, trazendo fortalecimento ao Java, sendo que IBM e Oracle sempre investiram e fizeram negócios através da plataforma Java. Nos dias de hoje, o Java se encontra na versão 8.



Figura 30 – Logotipo da empresa Oracle

Fonte: Oracle, 2014

* + 1. COMPONENTES DO JAVA

Segundo Caelum (2014a) os principais componentes para que seja possível o desenvolvimento de aplicações Java são: JVM, JRE (Java Runtime Environment) e JDK (Java Development Kit).

A JVM é a responsável para fazer a comunicação entre a aplicação e o sistema operacional.

O JRE é o ambiente de execução Java, composto pela JVM e bibliotecas, que são os itens necessários para fazer com que uma aplicação seja executada.

Por fim, o JDK é um conjunto de utilitários para o desenvolvimento em Java, e é composto pelo JRE e por algumas ferramentas, dentre elas o compilador.

* + 1. JAVA VITUAL MACHINE (JVM)

Como dito anteriormente, o Java foi desenvolvido para facilitar o desenvolvimento para diversas plataformas, e a maneira adotada para tornar isso possível foi fazer com que o código fosse compilado para rodar sobre a JVM (*Java Virtual Machine*) ao invés de ser compilado para rodar diretamente sobre o sistema operacional, como acontece em outras linguagens, como C e Pascal. Segundo Caelum (2014a) nessas linguagens, após o processo de compilação do código fonte, é gerado o código de máquina correspondente especificamente para uma plataforma, como exemplificado na Figura 31.



Figura 31 – Processo de compilação em linguagem C

Fonte: Caelum, 2014a

Esse código de máquina, conhecido como código binário, precisa se comunicar diretamente com o sistema operacional, e por isso, após compilado, ele somente conseguirá rodar com sucesso na plataforma determinada, conforme mostrado na Figura 32.



Figura 32 – Comunicação do código binário diretamente com o sistema operacional

Fonte: Caelum, 2014a

Em Java, esse processo é diferente pois a linguagem utiliza o conceito de máquina virtual. Segundo Caelum (2014a) isso significa que existe uma camada intermediária entre a aplicação e o sistema operacional, e essa camada se torna responsável por “traduzir” o código e efetuar as chamadas necessárias para cada sistema operacional permitindo que o desenvolvedor se abstraia dessa questão.

A JVM não trabalha diretamente com a linguagem Java, é necessário primeiro fazer a compilação do código escrito em Java, e esse processo irá gerar um código de máquina conhecido como *bytecode*, que é entendido pela máquina virtual. Após isso, a máquina virtual se tornará a responsável por interpretar esse código e comunicar-se com o sistema operacional, executando as instruções, como mostrado na Figura 33.



Figura 33 – Processo de comunicação entre aplicação e sistema operacional através da JVM.

Fonte: Caelum, 2014a

Segundo Caelum (2014a) a JVM utiliza o modelo de compilação JIT, onde o código é compilado em tempo real, durante a execução do programa, o que teoricamente pode trazer um pequeno ganho de desempenho, pois se a JVM encontrar algum ponto durante a execução que não está apresentando o desempenho esperado, ela pode tentar otimizar apenas esse trecho, ou tentar mudar a estratégia de otimização.

As Figuras Figura 34 e Figura 35 apresentam um código escrito em Java e o seu bytecode, respectivamente.



Figura 34 – Exemplo de código escrito em Java

Fonte: Caelum, 2014a



Figura 35 – Bytecode gerado após a compilação do código mostrado na

Fonte: Caelum, 2014a

* + 1. IDEs

As principais IDEs gratuitas para o desenvolvimento em Java são o Eclipse e o Netbeans. Segundo Caelum (2014b) a Eclipse atualmente é a IDE líder de mercado, e se encontra na versão 4.4. Ela possui seu código livre e é mantida por um consórcio de empresas liderado pela IBM. Embora ela seja mais utilizada para o desenvolvimento em Java, ela oferece suporte para outras linguagens como C, C++ e PHP através de plug-ins que podem ser baixados e incorporados à IDE.

Já a NetBeans começou a ser desenvolvida em 1996 sob o nome de Xelfi. Segundo o site da própria Netbeans (2014a) o objetivo era desenvolver uma IDE para desenvolvimento em Java escrita nessa mesma linguagem. Em 1999 a Sun Microsystems adquiriu a companhia que levava o mesmo nome da IDE, NetBeans, e em 2000 tornou seu código público. Atualmente na versão 8.0.1 e mantida pela Oracle, também apresenta suporte para desenvolvimento em outras linguagens como C, C++, PHP e Javascript.

As Figuras Figura 36 e Figura 37 apresentam o ambiente de desenvolvimento das IDEs Eclipse e NetBeans, respectivamente.



Figura 36 – Ambiente de desenvolvimento da Eclipse

Fonte: Eclipse, 2014



Figura 37 – Ambiente de desenvolvimento da NetBeans

Fonte: Netbeans, 2014b

* + 1. INTERFACE GRÁFICA EM JAVA

Segundo apostila da Caelum (2014c) existem atualmente duas bibliotecas gráficas que são oficialmente suportadas pelo Java, sendo elas AWT e Swing. A AWT (*Abstract Window Toolkit*) foi a primeira delas a surgir, sendo posteriormente superada pela Swing, na versão 1.2 do Java, que apresenta algumas vantagens sobre o antecessor.

Ainda segundo Caelum (2014c) essas duas bibliotecas são as oficiais, distribuídas em JREs e JDKs, porém existem outras APIs para desenvolvimento de interfaces gráficas de terceiros, sendo uma das mais famosas a desenvolvida pela IBM, a SWT (*Standard Widget Toolkit)*.

A principal vantagem da utilização das APIs gráficas do Java é a portabilidade. Segundo Caelum (2014c) o *look-and-feel* da biblioteca Swing é única, independente do sistema operacional sobre o qual a aplicação rodar ou qual a definição da tela do dispositivo.

O *look-and-feel* nada mais é segundo apostila da Caelum (2014c) do que o nome que leva a aparência da aplicação em si, como formatos, posicionamento de componentes, cores, etc. O próprio Java possui seu *look-and-feel* nativo, que se comporta da mesma maneira em qualquer plataforma desejada, porém é possível baixar *look-and-feels* de terceiros ou até mesmo desenvolver um por conta própria.

A Figura 38 apresenta diferentes *look-and-feels* para uma mesma aplicação.



Figura 38 – Utilização de diferentes *look-and-feels* para uma mesma aplicação

Fonte: Caelum, 2014c

1. ESTUDO DE CASO
   1. INTRODUÇÃO

Neste capítulo, serão apresentados os dados obtidos através do desenvolvimento do protótipo, assim como informações sobre o mesmo, incluindo trechos de código-fonte, diagramas UML e detalhes técnicos sobre a tecnologia escolhida e métodos de desenvolvimento.

O estudo de caso deste trabalho foi demonstrar a praticidade, funcionalidade e viabilidade do desenvolvimento multiplataforma ao desenvolver um protótipo de troca de mensagens utilizando o *framework* Qt. A aplicação é composta de dois componentes: O cliente (QTCC, detalhado no tópico 4.2), responsável por fazer a troca de mensagens entre diferentes usuários, independentemente da plataforma, e o servidor (QTCC\_Server, detalhado no tópico 4.4), responsável por gerenciar e servir de intermédio para todo o tráfego de mensagens entre os clientes, além de acesso ao banco de dados para persistência dos cadastros.



Figura 39 – Modelo de Arquitetura Cliente-Servidor da aplicação

Fonte: Própria

* 1. TECNOLOGIAS UTILIZADAS
     1. SQL SERVER
     2. GITHUB
  2. APLICAÇÃO CLIENTE (QTCC)

A aplicação cliente foi chamada QTCC, uma alusão ao Trabalho de Conclusão de Curso (TCC) e ao uso do *framework* Qt. Como descrito no tópico 2.2, o uso desse *framework* se deveu à sua fama mundial, e por ser um dos principais nomes em *frameworks* para desenvolvimento multiplataforma.

O intuito da aplicação cliente é de realizar *login* de usuários, cadastrar novos usuários, adicionar contatos à conta de usuário, iniciar e gerenciar conversas, e realizar a troca de mensagens entre diferentes usuários. Tudo utilizando o mesmo código-fonte.

Logo no começo, foi constatada uma dificuldade que acompanhou todo o andamento do projeto: A linguagem de programação. O Qt é um *framework* baseado em C++, e até então não havia sido ensinada tal linguagem de programação no decorrer do curso. Apenas durante os últimos meses do último semestre é que foi apresentada, para todos os alunos da sala, a linguagem C++ e o *framework* Qt.

De modo a compensar tal deficiência de conteúdo, a partir do mês de julho foi instalada a IDE Qt Creator (descrita no tópico 2.4) em computador próprio, além da obtenção de uma apostila, proveniente do próprio site do *framework*, explicando os conceitos do desenvolvimento de aplicações utilizando Qt, contendo programas de exemplo para cada funcionalidade.

O desenvolvimento da aplicação cliente começou em meados de outubro, e durante seu desenvolvimento foram gerados diversos programas-teste utilizando Qt, cada qual testando funcionalidades consideradas primordiais na aplicação (como comunicação por rede e serialização e deserialização de objetos em JSON).

* + 1. INTERFACE DA APLICAÇÃO

Como explicado nos tópicos 2.4.1 e 2.4.2, existem dois modos de se desenhar a interface de usuário em uma aplicação *desktop* utilizando a IDE Qt Creator: Utilizando Qt Quick e utilizando Qt Widgets.

Ao longo do curso, as disciplinas voltadas à programação ensinaram a desenhar as UIs no modo tradicional, mais voltado para aplicações *desktop*, com componentes possuindo tamanhos previamente especificados e formatos padronizados, como na Figura 40:



Figura 40 – Exemplo de interface de uma aplicação *desktop*

Fonte: Blanchette; Summerfield, 2008

Devido a essa proximidade com o modo no qual se foi ensinado a desenhar interfaces de usuário, foi escolhido o módulo Qt Widgets para a criação das telas da aplicação cliente, pois sua curva de aprendizado se apresentava menor do que aprender por completo o modo de programação declarativa do módulo Qt Quick, mais especificamente de sua principal linguagem, o QML.

Mesmo assim, houveram diversos momentos onde foi necessário reaprender a desenhar interfaces de usuário, em sua grande parte devido ao modo como o *framework* Qt funciona.

O uso de *layout managers* (visto dentro do tópico 2.4.2) permite que a interface, e seus componentes, tenham suas dimensões automaticamente reformuladas, de acordo com o tamanho da janela da aplicação. Isto se provou particularmente útil ao projeto, visto que a aplicação também será executada em plataformas *mobile*. Os smartphones possuem diversas resoluções de tela, dependendo do aparelho e marca, e todos os aplicativos *mobile* são executados em tela cheia. Utilizando *layouts*, foi possível fazer com que os componentes se escalonassem na tela, seja em plataformas *desktop* ou *mobile*. A Figura 41 exemplifica o resultado do uso de *layout managers* para o protótipo da tela de mensagens. Na imagem, foram utilizados os S.Os Android e Windows, respectivamente:



Figura 41 – Protótipo da tela de mensagens na plataforma Android e Windows

Fonte: Própria

Outro problema detectado durante o desenvolvimento das interfaces gráficas da aplicação foi sobre os *dialogs* (ou caixa de diálogo, em tradução livre). As caixas de diálogo tem, por padrão, o comportamento de impedir a ação do usuário na tela que a invocou, o que as torna comum para telas de *login*, cadastro e pesquisa.

Quando a aplicação cliente é executada em ambientes *desktop*, a exibição ocorreu normalmente. Entretanto, quando executado em um ambiente *mobile*, a caixa de diálogo apresentou problemas de exibição. Ela não é exibida em tela cheia, além de não bloquear o acesso a tela anterior, como visto na Figura 42. Até então não foi encontrada uma solução rápida para o problema, a não ser utilizar janelas normais.



Figura 42 – Exemplo de comportamento ocorrido ao utilizar *dialog* em ambiente Android

Fonte: Própria

Também foram encontrados problemas para abrir arquivos nas plataformas *mobile*. A caixa de diálogo para abertura de arquivos não é exibida de forma nativa nestas plataformas. Um exemplo desse comportamento pode ser visto na Figura 43:



Figura 43 – Exemplo de caixa de diálogo de abertura de arquivo no Android

Fonte: Própria

Não foi encontrada uma solução para o problema, visto que, para exibir a caixa de diálogo para abrir arquivos utilizando Qt Widgets, a classe responsável precisa necessariamente ser esta.

* + 1. SERIALIZAÇÃO EM JSON

Os objetos e entidades dos tipos usuário, mensagem, login, entre outros, pertencem às classes especificadas dentro da pasta “vo” (Value Object), como pode ser visto na Figura 44:



Figura 44 – Classes contidas na pasta "vo"

Fonte: Própria

Todas essas classes, exceto “Constantes” (classe contendo os enumeradores que identificam o tipo de pacote transferido entre cliente e servidor, o tipo de mensagem enviada/recebida, e o status de envio da mensagem), são, durante o fluxo da aplicação, utilizadas para transmitir via rede (vide tópico 4.3.3) os objetos ao qual cada classe se refere, seja uma nova mensagem, um novo cadastro, ou a busca por um contato. Para transmitir tais objetos, inclusive entre plataformas e linguagens de programação diferentes, é necessário que o objeto seja serializado na origem e, ao chegar no destino, seja deserializado, mantendo toda a integridade dos dados.

A serialização e deserialização de um objeto consiste em transformar os dados binários deste em um vetor de bytes e vice-versa, possibilitando, por exemplo, o envio e recebimento de um objeto pela rede, assim como a gravação do objeto em arquivo no disco rígido, e, respectivamente, sua leitura. Na aplicação cliente, as mensagens trocadas não são salvas no banco de dados, fazendo seu armazenamento se basear no disco rígido da máquina que estiver executando a aplicação. Além das conversas, é salva localmente uma cópia da lista de contatos do usuário, de modo a economizar o tráfego de rede. Para realizar a função de serialização/deserialização dos dados, o padrão escolhido desde o início do projeto foi o JSON.

O JSON (JavaScript Object Notation), segundo o próprio site do padrão (JSON, 2014), é “um formato de texto completamente independente de linguagem (de programação)”. Ele foi criado com base no JavaScript e é considerado uma das principais formas de envio e persistência de objetos.

Os objetos JSON, segundo JSON (2014), são compostos de dois tipos de elementos:

* Pares propriedade-valor, muito semelhante às propriedades do QML (vide tópico 2.4.1.1).
* Uma lista ordenada de valores. Um vetor contendo diversos sub-objetos.

O *framework* Qt possui um conjunto de classes voltadas especificamente para a serialização de objetos em JSON. Essas classes são:

* QJsonObject
  + É a representação, em JSON, de um objeto. Permite armazenar valores, vetores (QJsonArray) e outros QJsonObject. Para acessar o valor de uma propriedade, é necessário buscar pelo identificador alfanumérico da desta.
* QJsonDocument
  + Permite a manipulação de um QJsonObject, como conversão de/para um vetor de *bytes*
* QJsonArray
  + Representa um vetor (*array*) de objetos (no caso, de QJsonObject)

O Quadro 1 exemplifica o uso de QJsonObject ao mostrar os métodos de serialização e deserialização de um objeto do tipo EContato. A subclasse Campos guarda os nomes dos campos em JSON referentes à cada propriedade da classe onde se encontra.

|  |
| --- |
| EContato EContato::Deserializar(QJsonObject &json)  {  EContato c;  c.\_id = json[EBase::Campos::ID].toInt();  c.\_nome = json[Campos::Nome].toString();  c.\_inativo = json[Campos::Inativo].toBool();  QImage a;  QByteArray array = json[Campos::Foto].toString().toLatin1();  if(a.loadFromData(QByteArray::fromBase64(array)))  c.\_foto = a;  return c;  }  QJsonObject EContato::Serializar(EContato c)  {  QJsonObject json;  json[EBase::Campos::ID] = c.\_id;  json[Campos::Nome] = c.\_nome;  QByteArray array;  QBuffer buffer(&array);  buffer.*open*(QIODevice::WriteOnly);  c.\_foto.save(&buffer,"PNG");  json[Campos::Foto] = QString(array.toBase64(QByteArray::Base64Encoding));  json[Campos::Inativo] = c.\_inativo;  return json;  } |

Quadro 1 – Métodos "Deserializar" e "Serializar" da classe EContato

Fonte: Própria

* + 1. ENVIO PELA REDE

A comunicação entre a aplicação cliente e o servidor é realizada através do protocolo de rede TCP/IP. Os motivos para tal escolha são o fato de que cliente e servidor foram feitos em linguagens de programação e IDEs diferentes, portanto deveria ser um protocolo padronizado, além de que, devido à natureza da aplicação (troca de mensagens), era necessário que houvesse, tanto por parte do servidor como do cliente, uma confirmação de recebimento do pacote completo.

O Qt possui, para tal comunicação, a classe QTcpSocket, que pode funcionar tanto como servidor quanto como cliente. Essa classe possui métodos para ler dados da rede, gravar, se conectar a hosts remotos, entre outros.

Na aplicação cliente, foi criado, dentro da pasta “util”, a classe ComunicacaoRede. Essa classe, que deve ser instanciada, possui um atributo chamado “socket”, da classe QTcpSocket, além do método enviaPacote.

O socket é responsável por estabelecer a comunicação com o servidor, além de servir de meio para a transferência dos pacotes. O método enviaPacote, que retorna o pacote recebido do servidor, serve para enviar o comando ou objeto pela rede até o servidor. O Quadro 2 mostra o funcionamento do método enviaPacote:

|  |
| --- |
| QString ComunicacaoRede::enviaPacote(QString pacote)  {  socket = new QTcpSocket(this);  while(!(socket->state()==QAbstractSocket::ConnectedState))  {  socket->*connectToHost*("localhost",5500);  while(!socket->*waitForConnected*(10000))  {  Logger::debug("não conseguiu");  }  }  QString retorno;  socket->write(pacote.toLatin1());  // socket->flush();  socket->*waitForBytesWritten*(-1);  socket->flush();  //while(!socket->bytesAvailable()>0)  while (socket->*waitForReadyRead*(-1))  {  retorno += QString(socket->readAll());  }  if(retorno.startsWith("Falha: "))  return "";  else  return retorno;  } |

Quadro 2 – Método enviaPacote da classe ComunicacaoRede

Fonte: Própria

De modo que o servidor identifique que tipo de pacote ele está recebendo, foi especificado um enumerador (EtiposPacotesDadosEnum) contendo uma lista de cabeçalhos padronizados para se adicionar ao pacote a ser enviado (OP Code). Esse enumerador se encontra-se no Quadro 3:

|  |
| --- |
| enum ETiposPacotesDadosEnum{  RequisicaoLogin = 1,  ReceberNovasMensagens=2,  EnviarNovaMensagem=3,  StatusContato=4,  StatusMensagem=5,  NovoCadastro=6,  EnviarNovoUsuario=7,  EnviarNovoGrupo=8,  BuscaUsuarioPeloEmail=9,  BuscaUsuarioPeloID=10  }; |

Quadro 3 – Enumerador EtiposPacotesDadosEnum

Fonte: Própria

* 1. APLICAÇÃO SERVIDOR (QTCC\_Server)

A aplicação servidor foi chamada QTCC\_Server. O sufixo “\_Server” foi dado de modo a diferenciá-la do cliente, mas mantendo a relação entre as duas aplicações.

A aplicação servidor tem a função primordial de gerenciar os acessos que cada cliente deseja fazer à base de dados do sistema.

A aplicação servidor, por rodar apenas em uma plataforma, não era necessária que fosse desenvolvida no *framework* Qt, portanto foi utilizado o Visual Studio 2013 para desenvolvê-la em C#.

* + 1. INTERFACE DA APLICAÇÃO
    2. SERIALIZAÇÃO EM JSON
    3. ENVIO PELA REDE
    4. PERSISTÊNCIA DOS DADOS

**Aqui se fala sobre todo o processo de desenvolvimento da aplicação. Quem for escrever o código-fonte provavelmente vai ser quem vai mexer nesse capítulo (até onde eu entendi)**

1. CONCLUSÃO

**A que resposta chegamos em relação à justificativa de pesquisa? Quais são as considerações finais sobre o desenvolvimento do protótipo, e, consequentemente, sobre o tema de pesquisa**

1. TRABALHOS FUTUROS

**Quais são as possíveis implementações que poderiam/deveriam (não falar que não deu por tempo) ter sido implementadas, e para que a pesquisa atual serviria de base em uma pesquisa futura (tese, talvez?)**

1. REFERÊNCIAS

BLANCHETTE, J.; SUMMERFIELD, M.; *C++ GUI Programming with Qt 4, Second Edition.* 2008.Disponível em <http://www.bogotobogo.com/cplusplus/files/c-gui-programming-with-qt-4-2ndedition.pdf>. Acesso em 31 agosto 2014.

CAELUM. *O que é Java.* 2014a. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/o-que-e-java/>. Acesso em 12 de novembro 2014.

\_\_\_\_\_\_. *Eclipse IDE.* 2014b. Disponível em: <http://www.caelum.com.br/apostila-java-orientacao-objetos/eclipse-ide/>. Acesso em 13 de novembro 2014.

\_\_\_\_\_\_. *Interfaces gráficas em Java.* 2014c. Disponível em: <http://www.caelum.com.br/apostila-java-testes-xml-design-patterns/interfaces-graficas-com-swing/>. Acesso em 15 de novembro 2014.

DEITEL, H.; DEITEL, P.; *Java How to Program, Ninth Edition.* 2012. Disponível em: <http://uet.vnu.edu.vn/~chauttm/e-books/java/JH2P-9th.pdf>. Acesso em 12 de novembro 2014.

DICIO. *Dicionário online de português*. 2014. Disponível em: <http://www.dicio.com.br>. Acesso em 19 novembro 2014.

ECLIPSE. *Eclipse Screenshots.* 2014. Disponível em: <http://www.eclipse.org/screenshots/images/SDK-RedFlag\_Linux.png>. Acesso em 14 de novembro 2014.

ECMA. *ECMA 404*. 2014. Disponível em: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>. Acesso em 19 novembro 2014.

EZUST, A; EZUST, E; *Introduction to Design Patterns in C++ with Qt, Second Edition.* 2011. Disponível em <http://it-ebooks.info/book/1301/>. Acesso em 26 outubro 2014.

FONSECA FILHO, C. *História da Computação: O caminho do pensamento e da tecnologia.* 2007. Disponível em: <http://www.pucrs.br/edipucrs/online/historiadacomputacao.pdf>. Acesso em 29 setembro 2014.

JAVA. *Obtenha informações sobre a Tecnologia Java.* 2014. Disponível em: <https://www.java.com/pt\_BR/about/>. Acesso em 16 novembro 2014.

JSON. *JSON*. 2014. Disponível em: <http://json.org/>. Acesso em 19 novembro 2014.

KDE. *KDE Project Announced*. 2014a. Disponível em: <http://www.kde.org/announcements/announcement.php>. Acesso em 28 setembro 2014.

\_\_\_\_\_\_*Plasma*. 2014b. Disponível em: <https://userbase.kde.org/Plasma>. Acesso em 28 setembro 2014.

\_\_\_\_\_\_*What is KDE*. 2014c. Disponível em: <https://userbase.kde.org/What\_is\_KDE>. Acesso em 28 setembro 2014.

\_\_\_\_\_\_*KDE Free Qt Foundation*. 2014d. Disponível em: <http://www.kde.org/community/whatiskde/kdefreeqtfoundation.php>. Acesso em 28 setembro 2014.

MICHAELIS. *Dicionário de Português Online.* 2014. Disponível em: <http://www.michaelis.uol.com.br>. Acesso em 19 novembro 2014.

MOLKENTIN, D. *Book of Qt4.* 2006. Disponível em <http://www-cs.ccny.cuny.edu/~wolberg/cs221/qt/books/BookOfQt4.pdf>. Acesso em 14 setembro 2014.

MONO. *About Mono*. 2014a. Disponível em: <http://www.mono-project.com/docs/about-mono>. Acesso em 30 outubro 2014.

\_\_\_\_\_\_*C# Compiler.* 2014b. Disponível em: <http://www.mono-project.com/docs/about-mono/languages/csharp>. Acesso em 31 outubro 2014.

\_\_\_\_\_\_*Ahead-of-Time Compilation (AOT)*. 2014c Disponível em: <http://www.mono-project.com/docs/advanced/runtime/docs/aot>. Acesso em 31 outubro 2014.

\_\_\_\_\_\_*Mono Runtime.* 2014d. Disponível em: <http://www.mono-project.com/docs/advanced/runtime>. Acesso em 31 outubro 2014.

\_\_\_\_\_\_*Supported Platforms.* 2014e. Disponível em: <http://www.mono-project.com/docs/about-mono/supported-platforms>. Acesso em 01 novembro 2014.

\_\_\_\_\_\_*MonoDevelop.* 2014f. Disponível em <http://monodevelop.com/>. Acesso em 01 novembro 2014.

NetBeans. *A Brief History of NetBeans.* 2014a. Disponível em: <https://netbeans.org/about/history.html>. Acesso em 14 novembro 2014.

\_\_\_\_\_\_. *Netbeans Images.* 2014b. Disponível em: <https://netbeans.org/images\_www/v7/1/screenshots/web-app.png>. Acesso em 14 novembro 2014.

Oracle. *Oracle Logos.* 2014. Disponível em: <http://www.oracle.com/us/corporate/press/016236>. Acesso em 15 novembro 2014.

QT DIGIA. *Qt Creator Enterprise*. 2014a. Disponível em: <http://qt.digia.com/Product>. Acesso em 05 setembro 2014.

\_\_\_\_\_\_*Qt 5.4 Alpha Available.* 2014b. Disponível em: <http://blog.qt.digia.com/blog/2014/09/08/qt-5-4-alpha-available/>. Acesso em 01 outubro 2014.

QT PROJECT. *Qt for WinRT*. 2014a. Disponível em: <http://qt-project.org/wiki/WinRT>. Acesso em 28 agosto 2014.

\_\_\_\_\_\_*Supported Platforms*. 2014b. Disponível em: <http://qt-project.org/doc/qt-5/supported-platforms.html>. Acesso em 31 agosto 2014.

\_\_\_\_\_\_*IDE Overview.* 2014c. Disponível em: <http://qt-project.org/doc/qtcreator-3.2/creator-overview.html>. Acesso em 31 agosto 2014.

\_\_\_\_\_\_*Qt Quick*. 2014d. Disponível em: <http://qt-project.org/doc/qt-5/qtquick-index.html>. Acesso em 01 setembro 2014.

\_\_\_\_\_\_*QML Applications*. 2014e Disponível em: <http://qt-project.org/doc/qt-5/qmlapplications.html>. Acesso em 01 setembro 2014.

\_\_\_\_\_\_*Qt Widgets.* 2014f. Disponível em: <http://qt-project.org/doc/qt-5/qtwidgets-index.html>. Acesso em 01 setembro 2014.

\_\_\_\_\_\_*New Features in Qt 5.3*. 2014g. Disponível em: <http://qt-project.org/wiki/New-Features-in-Qt-5.3>. Acesso em 05 setembro 2014.

\_\_\_\_\_\_*Qt QML*. 2014h. Disponível em: <http://qt-project.org/doc/qt-5/qtqml-index.html>. Acesso em 19 setembro 2014.

\_\_\_\_\_\_*Qt Widgets C++ Classes.* 2014i. Disponível em: <http://qt-project.org/doc/qt-5/qtwidgets-module.html>. Acesso em 30 setembro 2014.

\_\_\_\_\_\_*User Interfaces.* 2014j. Disponível em: <http://qt-project.org/doc/qt-5/topics-ui.html>. Acesso em 30 setembro 2014.

\_\_\_\_\_\_*Qt WebKit.* 2014k. Disponível em: <http://qt-project.org/doc/qt-5/qtwebkit-index.html>. Acesso em 30 setembro 2014.

\_\_\_\_\_\_*Whats’s New in Qt 4.* 2014l. Disponível em: <http://qt-project.org/doc/qt-4.8/qt4-intro.html>. Acesso em 30 setembro 2014.

\_\_\_\_\_\_*What’s New in Qt 5.* 2014m. Disponível em: <http://qt-project.org/doc/qt-5/qt5-intro.html>. Acesso em 30 setembro 2014.

\_\_\_\_\_\_*Layout Management*. 2014n. Disponível em: <http://qt-project.org/doc/qt-5/layout.html>. Acesso em 06 outubro 2014.

\_\_\_\_\_\_*Signals & Slots.* 2014o. Disponível em: <http://qt-project.org/doc/qt-4.8/signalsandslots.html>. Acesso em 06 outubro 2014.

\_\_\_\_\_\_*New\_Signal\_Slot\_Syntax.* 2014p. Disponível em: <http://qt-project.org/wiki/New\_Signal\_Slot\_Syntax>. Acesso em 26 outubro 2014.

ROSA, R. E. V. S.; GIL, A. M.; MENDONÇA, P. R. B.; COSTA FILHO, C. F. F.; LUCENA JR., V. F.; *Desenvolvimento Rápido de Aplicações Móveis Utilizando a Linguagem Declarativa QML.* 2011. Disponível em: <http://www.die.ufpi.br/ercemapi2011/minicursos/MC10.pdf>. Acesso em 15 setembro 2014.

THELIN, J. *Foundations of Qt Development.* 2007. Disponível em: <https://raw.githubusercontent.com/bharathwaaj/sandbox/master/Programming%20books/qt/Foundations.of.Qt.Development.pdf>. Acesso em 26 outubro 2014.

WIRESHARK. *QtShark*. 2014a. Disponível em: <http://wiki.wireshark.org/Development/QtShark>. Acesso em 15 setembro 2014.

\_\_\_\_\_\_*User’s Guide.* 2014b. Disponível em: <https://www.wireshark.org/download/docs/user-guide-us.pdf>. Acesso em 28 setembro 2014.

\_\_\_\_\_\_*About.* 2014c. Disponível em: <https://www.wireshark.org/about.html>. Acesso em 28 setembro 2014.

\_\_\_\_\_\_*We’re switching to Qt.* 2014d. Disponível em: <https://blog.wireshark.org/2013/10/switching-to-qt/>. Acesso em 28 setembro 2014.

XAMARIN. *About Xamarin.* 2014a. Disponível em: <http://xamarin.com/about>. Acesso em 19 outubro 2014.

\_\_\_\_\_\_*Platform.* 2014b. Disponível em: <http://xamarin.com/platform>. Acesso em: 19 outubro 2014.

\_\_\_\_\_\_*Xamarin.Forms.* 2014c. Disponível em: <http://xamarin.com/forms>. Acesso em 19 outubro 2014.

\_\_\_\_\_\_*Installing Xamarin.iOS on Windows.* 2014d. Disponível em: <http://developer.xamarin.com/guides/ios/getting\_started/installation/windows/>. Acesso em 19 outubro 2014.

\_\_\_\_\_\_*Introduction do Xamarin Studio.* 2014e. Disponível em: <http://xamarin.com/studio>. Acesso em 19 outubro 2014.

\_\_\_\_\_\_*Xamarin’s Customers.* 2014f. Disponível em: <http://xamarin.com/customers>. Acesso em 20 outubro 2014.

\_\_\_\_\_\_*Introduction to Xamarin Test Cloud.* 2014g. Disponível em: <http://developer.xamarin.com/guides/testcloud/introduction-to-test-cloud/>. Acesso em 20 outubro 2014.

\_\_\_\_\_\_*Xamarin Test Cloud.* 2014h. Disponível em: <http://xamarin.com/test-cloud>. Acesso em 20 outubro 2014.

1. Multiplataforma: O conceito de multiplataforma se baseia em uma mesma aplicação (um mesmo código-fonte) ser compilável para diversos sistemas operacionais, independente de sua arquitetura e bibliotecas específicas. (Michaelis, 2014) [↑](#footnote-ref-1)
2. Plataforma: um ambiente (normalmente um sistema operacional) onde diversos programas são executados. Exemplos: Windows, Mac OS X, Linux, Android, iOS e Windows Phone. (Michaelis, 2014) [↑](#footnote-ref-2)
3. *Cross-platform:* Vide multiplataforma na página 15 [↑](#footnote-ref-3)