

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 Termomecânica 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 Termomecânica 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 insucesso é apenas uma  
oportunidade para recomeçar  
de novo com mais inteligência”*

*Henry Ford*

## RESUMO

Com o passar dos anos, o progresso tecnológico possibilitou a criação de 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 uma determinada plataforma funcione com falhas ou não funcione nas 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:** Desenvolvimento Multiplataforma. Qt. Troca de Mensagens

## **ABSTRACT**

As 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 a specific platform 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 Development. Qt. Message exchange

## LISTA DE QUADROS

Quadro 1 – Métodos "Deserializar" e "Serializar" da classe EContato .....	81
Quadro 2 – Método enviaPacote da classe ComunicacaoRede.....	82
Quadro 3 – Enumerador EtiposPacotesDadosEnum.....	83
Quadro 4 – Propriedade Foto e Foto_String .....	89
Quadro 5 – Classe JSON_Logic .....	90
Quadro 6 – Criação da <i>thread</i> tEscutaClientes .....	92
Quadro 7 – Método TrataPacote .....	93
Quadro 8 – Método CriarBDXML .....	97
Quadro 9 – Método estático AdicionaContatoListContatos .....	98



## LISTA DE FIGURAS

Figura 1 – Logo do Qt .....	19
Figura 2 – Programas que utilizam Qt.....	23
Figura 3 – Exemplo de arquivo .pro .....	24
Figura 4 – Exemplo de classe com declarações de <i>signals</i> e <i>slots</i> .....	27
Figura 5 – Implementação de um <i>slot</i> com emissão de <i>signal</i> .....	28
Figura 6 – Exemplo do mecanismo de <i>signal</i> e <i>slot</i> .....	29
Figura 7 – Exemplo de código-fonte com conexão entre objetos utilizando <i>signals</i> e <i>slots</i> .....	30
Figura 8 – Tela inicial do Qt Creator.....	31
Figura 9 – Exemplo de propriedades atreladas à outras ( <i>property bindings</i> ).....	33
Figura 10 – Alguns tipos utilizados no sistema de tipagem da linguagem QML.....	33
Figura 11 – Exemplo de código em QML para criação de um retângulo .....	34
Figura 12 – Exemplo de interface gráfica feita com o módulo Qt Widgets .....	35
Figura 13 – Exemplo de Estilos dos <i>Widgets</i> .....	36
Figura 14 – Exemplos de gerenciadores de <i>layouts</i> do módulo Qt Widgets .....	37
Figura 15 – Wireshark 2 <i>Preview</i> em funcionamento .....	38
Figura 16 – A comunidade KDE .....	39
Figura 17 – Interface do Plasma.....	40
Figura 18 – Logo da empresa Xamarin .....	43
Figura 19 – Código para a criação de uma tela de <i>login</i> comum .....	44
Figura 20 – Código mostrado na Figura 19 rodando, respectivamente, em iOS, Android e Windows Phone .....	44
Figura 21 – Criação de interface para iOS utilizando-se o Xamarin para Visual Studio .....	46
Figura 22 – Criação de interface para iOS utilizando-se o Xamarin Studio.....	47
Figura 23 – Software Rdio rodando em iOS ( <i>tablet</i> ) e Android ( <i>smartphone</i> ).....	48
Figura 24 – Aplicativo da empresa Kimberly-Clark desenvolvido com utilização do Xamarin .....	49
Figura 25 – Interface do Xamarin Test Cloud .....	51
Figura 26 – Logotipo da Mono .....	52
Figura 27 – Plataformas e arquiteturas suportadas.....	54
Figura 28 – Tela principal do MonoDevelop .....	57
Figura 29 – Tela de criação de interfaces da IDE MonoDevelop.....	58
Figura 30 – Logotipo do Java.....	59
Figura 31 – Logotipo da empresa Oracle .....	60
Figura 32 – Processo de compilação em linguagem C .....	61
Figura 33 – Comunicação do código binário diretamente com o sistema operacional.....	61
Figura 34 – Processo de comunicação entre aplicação e sistema operacional através da JVM.....	62
Figura 35 – Exemplo de código escrito em Java.....	63
Figura 36 – <i>Bytecode</i> gerado após a compilação do código mostrado na Figura 35 .....	63
Figura 37 – Ambiente de desenvolvimento da IDE Eclipse .....	64
Figura 38 – Ambiente de desenvolvimento do NetBeans.....	65
Figura 39 – Utilização de diferentes <i>look-and-feels</i> para uma mesma aplicação.....	66
Figura 40 – Modelo de Arquitetura Cliente-Servidor da aplicação.....	69
Figura 41 – Tela inicial do Git Bash.....	70
Figura 42 – Tela inicial do Git GUI .....	71
Figura 43 – Visualizando alterações no repositório através do Git.....	72
Figura 44 – Exemplo de interface de uma aplicação <i>desktop</i> .....	74
Figura 45 – Protótipo da tela de mensagens na plataforma Android (a), Windows (b) e Linux (c).....	75

Figura 46 – Exemplo de comportamento ocorrido ao utilizar <i>dialog</i> em ambiente Android.....	76
Figura 47 – Exemplo de caixa de diálogo de abertura de arquivo no Android .....	77
Figura 48 – Classes contidas na pasta "vo" .....	78
Figura 49 – Propriedade em JSON.....	79
Figura 50 – Vetor em JSON.....	79
Figura 51 – Tela Principal da aplicação servidor .....	84
Figura 52 – Tela principal da aplicação servidor sem selecionar um usuário .....	85
Figura 53 – Selecionando para adicionar uma referência ao projeto em C#.....	87
Figura 54 – Adicionando uma referência ao projeto .....	88
Figura 55 – Diagrama Entidade-Relacionamento (DER) do banco QTCC .....	94
Figura 56 – Tela de reconfigurar acesso à base de dados.....	95
Figura 57 – Mensagem de erro ao realizar configuração incorreta de conexão .....	95
Figura 58 – Diagrama de classes da aplicação CFG_BD_XML .....	96

## LISTA DE SIGLAS

**AOT** – Ahead-of-Time

**API** – Application Programming Interface

**AWT** – Abstract Window Toolkit

**CSS** – Cascading Style Sheets

**DAO** – Data Access Object

**DER** – Diagrama Entidade-Relacionamento

**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

**JSON** – JavaScript *Object Notation*

**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

**SGBD** – Sistema Gerenciador de Banco de Dados

**SO** – Sistema Operacional

**SQL** – Structured Query Language

**SVG** – Scalable Vector Graphics

**SWT** – Standard Widget Toolkit

**TCP** – Transmission Control Protocol

**UI** – User Interface

**VO** – Value Object

**WinRT** – Windows Runtime

**XML** – eXtended Markup Language

# SUMÁRIO

1	INTRODUÇÃO .....	14
1.1	Objetivo Geral .....	16
1.2	Objetivos Específicos.....	16
1.3	Justificativa .....	16
1.4	Metodologia .....	17
1.5	Estrutura do Trabalho .....	17
2	TECNOLOGIA UTILIZADA .....	19
2.1	Introdução.....	19
2.2	Qt.....	19
2.2.1	História do Qt .....	19
2.2.1.1	Qt 4.....	21
2.2.1.2	Qt 5.....	22
2.2.2	Compilação multiplataforma.....	23
2.2.3	<i>Signals and slots</i> .....	26
2.3	Qt Creator.....	31
2.3.1	Qt Quick.....	32
2.3.1.1	QML .....	32
2.3.2	Qt Widgets.....	34
2.4	Exemplos de Aplicações que Utilizam Qt .....	37
2.4.1	Wireshark .....	37
2.4.2	KDE .....	39
3	ALTERNATIVAS AO QT .....	42
3.1	Introdução .....	42
3.2	Xamarin .....	42
3.2.1	Xamarin.Forms .....	43
3.2.2	IDEs.....	45
3.2.3	Clientes da Xamarin.....	47
3.2.4	Xamarin Test Cloud .....	49
3.2.5	Vantagens e desvantagens.....	51
3.3	Mono .....	52
3.3.1	Componentes da Mono.....	53
3.3.2	Sistemas operacionais suportados.....	54
3.3.3	Benefícios .....	55

3.3.4	IDE .....	56
3.3.5	Vantagens e desvantagens.....	58
3.4	Java.....	59
3.4.1	História do Java .....	59
3.4.2	Componentes do Java .....	60
3.4.3	Java Virtual Machine (JVM) .....	61
3.4.4	IDEs.....	64
3.4.5	Interface gráfica em Java.....	65
3.4.6	Vantagens e desvantagens.....	66
4	ESTUDO DE CASO .....	68
4.1	Introdução .....	68
4.2	Tecnologias Utilizadas .....	69
4.2.1	SQL Server .....	69
4.2.2	Git.....	69
4.3	Delimitações do Projeto.....	72
4.4	Aplicação Cliente (QTCC).....	73
4.4.1	Interface da aplicação .....	73
4.4.2	Serialização em JSON .....	77
4.4.3	Envio pela rede.....	81
4.5	Aplicação Servidor (QTCC_Server) .....	83
4.5.1	Interface da aplicação .....	83
4.5.2	Serialização em JSON .....	86
4.5.3	Troca de dados pela rede.....	91
4.5.4	Persistência dos dados .....	93
5	CONCLUSÃO .....	99
6	TRABALHOS FUTUROS.....	100
	REFERÊNCIAS.....	101
	APÊNDICE A: PROTÓTIPO DA APLICAÇÃO .....	107

## 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 S.O. 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 Macintosh, 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<sup>1</sup>. Cada S.O. 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<sup>2</sup>”, 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 *Integrated Development Environment* (IDE) Qt Creator, que, utilizando compiladores

---

<sup>1</sup> 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)

<sup>2</sup> Multiplataforma: O conceito de multiplataforma se baseia em uma mesma aplicação (um mesmo código-fonte) ser compilável para diversos sistemas operacionais, independentemente de sua arquitetura e bibliotecas específicas. (Michaelis, 2014)



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.

## 1.1 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.

## 1.2 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.

## 1.3 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 S.O. 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++.

Utilizado em mais de 70 indústrias e em milhões de dispositivos (QT IO, 2014b), o *framework* Qt pareceu ser uma ferramenta robusta e repleta de funcionalidades. Portanto, o atual estudo visa verificar a utilidade e viabilidade em utilizá-la para uma aplicação cliente-servidor com comunicação TCP.

## 1.4 Metodologia

Levantamento bibliográfico em livros sobre assuntos ligados ao tema, e documentação própria dos mantenedores do *framework* Qt, assim como um breve estudo bibliográfico sobre plataformas concorrentes, sendo elas Xamarin, Mono e Java.

Estudo de Caso, através do desenvolvimento de um protótipo de aplicação para trocas de mensagens de texto entre clientes rodando em diferentes plataformas. Para os testes da aplicação, foram escolhidas inicialmente as plataformas Windows (*desktop*) e Android (*mobile*) por serem as mais utilizadas.

A cada nova funcionalidade implementada, o protótipo foi recompilado para ambas as plataformas e foi analisado seu comportamento. Caso houvesse alguma irregularidade ou diferença de uma plataforma para outra, o trecho era reescrito, de maneira que funcionasse da mesma maneira em ambas as plataformas, sem comprometer partes já funcionais do protótipo.

## 1.5 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.

## 2 TECNOLOGIA UTILIZADA

### 2.1 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.

### 2.2 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 IO, 2014c

#### 2.2.1 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, Macintosh 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. Em 1994, eles fundaram a Quasar Technologies, que eventualmente se transformaria em Trolltech.

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 2.4.2), 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 o consolidou como a principal ferramenta de 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.

#### 2.2.1.1 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, 2014k).

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 (2014k), em 2006 a versão 4.2 trouxe o suporte ao Windows Vista e suporte à *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 à *eXtended Markup Language* (XML) e a compatibilidade com o Windows CE, sistema operacional da Microsoft para *tablets* e dispositivos móveis na época. Como visto em Qt Digia (2014a), ainda em 2008, a Trolltech, junto com o *framework* Qt, foi adquirida pela Nokia, mantendo a disponibilidade *open-source* de sua ferramenta.

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 Macintosh de 64-bits (QT PROJECT, 2014k).

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 grande atualização da versão 4 do Qt, a 4.8, onde basicamente foram trazidas melhorias de performance (QT PROJECT, 2014k). Esta versão continua sendo suportada atualmente, de modo a facilitar a migração das aplicações compiladas na versão 4 do *framework* para a mais atual (QT DIGIA, 2014c).

Segundo dados de Qt Digia (2014b), no começo de 2012 a empresa Digia iniciou o processo de aquisição dos direitos sobre o *framework* Qt, sendo sua atual proprietária.

### 2.2.1.2 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 (2014l), 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 versão 5.3, focada em desempenho, estabilidade e usabilidade, além do suporte experimental para o Windows Runtime e a plataforma Windows Phone (QT PROJECT, 2014l).

Em dezembro de 2014 foi lançada a versão 5.4 do *framework*, e as principais novidades são o suporte total à interface Windows RT, melhoria de motores gráficos e o suporte para que os componentes tenham a aparência nativa do Android (QT IO, 2014c).

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

### 2.2.2 Compilação multiplataforma

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 (QT PROJECT, 2014b). Segundo o Qt IO (2014c), a partir da versão 5.4 foi implementado 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.



Para realizar esta compilação multiplataforma, o *framework* Qt faz uso dos arquivos de projeto (com extensão .pro) e Makefile. Esses arquivos podem ser criados por uma ferramenta própria de compilação do Qt, o qmake, ou por programas de terceiros, como CMake e Boost, e são arquivos independentes de plataforma (BLANCHETTE; SUMMERFIELD, 2008)

Como visto em Blanchette e Summerfield (2008) e Molkentin (2006), o arquivo de projeto possui informações referentes aos arquivos e variáveis de sistema da aplicação, como arquivos de código-fonte, definições de nome da aplicação, tipo de aplicação gerada (se é um programa executável ou uma biblioteca), assim como configurações de módulos a utilizar ou remover.

Utilizando a ferramenta qmake, o arquivo .pro pode ser criado na linha de comando, apontando para a pasta do projeto, com o seguinte comando:

#### **qmake –project**

A Figura 3 mostra um exemplo de arquivo de projeto criado para a aplicação HelloWorld:

```
#helloWorld/helloWorld.pro

#####
# Automatically generated by qmake
#####

TEMPLATE = app
CONFIG -= moc
DEPENDPATH += .
INCLUDEPATH += .

# Input
SOURCES += main.cpp
```

Figura 3 – Exemplo de arquivo .pro  
Fonte: Molkentin, 2006

O arquivo Makefile é gerado a partir de um arquivo .pro, e contém informações necessárias para que a aplicação seja compilada na plataforma atual, através dos programas make ou nmake (exclusivo para compilar com base na IDE Visual Studio).

Para gerar o arquivo Makefile através da ferramenta qmake, é possível executar em linha de comando, na pasta onde se encontra o arquivo .pro, o seguinte comando:

**qmake <nome do arquivo .pro>**

No Windows, são gerados três arquivos: Makefile, Makefile.Debug e Makefile.Release, onde o primeiro arquivo serve como referência para os outros dois. Em sistemas Unix e no Mac OS X é gerado apenas o arquivo Makefile (MOLKENTIN, 2006).

Blanchette e Summerfield (2008) declaram que é possível, nas distribuições Unix e no Mac OS X, especificar o conjunto de plataforma/compilador a ser utilizado, através do argumento `-spec`, seguido da combinação. O comando abaixo gera um arquivo Makefile, com base no arquivo `hello.pro`, utilizando o Intel C++ Compiler da distribuição Linux de 64 bits:

**qmake -spec linux-icc-64 hello.pro**

A listagem de especificações de combinações para o argumento `-spec` se encontram na pasta `mkspecs`, dentro da pasta de cada compilador utilizado no Qt (BLANCHETTE; SUMMERFIELD, 2008).

Com o arquivo Makefile gerado, basta executar o comando `make` ou `nmake` para compilar o projeto.

Outro aspecto que facilita na compilação de um mesmo projeto para diversos sistemas operacionais é o uso de diferentes compiladores. Durante a instalação do *framework* Qt, é possível escolher quais compiladores que devem estar disponíveis para as aplicações. Dependendo do S.O. em questão, certos compiladores estarão disponíveis para escolha, como o MinGW, que é um compilador *open-source*, e o Microsoft Visual C++, exclusivo para Windows.

Estes compiladores é que buscarão as bibliotecas nativas do sistema operacional onde estão sendo executados, e que gerarão o arquivo final da aplicação, seja este um arquivo executável ou uma biblioteca. Desse modo, a aplicação estará, onde estiver sendo executada, utilizando o código nativo da plataforma-alvo, garantindo seu desempenho.

A única desvantagem neste processo é de que, para se compilar uma aplicação para Mac OS X, por exemplo, é necessário ter o Qt instalado em uma máquina com esse sistema operacional e compilar o projeto nesta máquina, e assim também para qualquer sistema operacional *desktop*.

A plataforma Android necessita da instalação de um conjunto de programas externos, disponíveis em Windows, Mac e Linux, para que se possa compilar uma aplicação neste sistema operacional, enquanto que a compilação para iOS e Windows Phone necessitam de uma máquina com o sistema operacional Mac OS X e Windows na versão 8.1, respectivamente (QT PROJECT, 2014b).

### 2.2.3 *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 (2014n), 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 4 mostra o cabeçalho (*header*) de uma classe de exemplo com definições de *signal* e *slot*:

```

#include <QString>
#include <QObject>
class MyClass : public QObject
{
    Q_OBJECT

public:
    MyClass( const QString &text, QObject *parent = 0 );

    const QString& text() const;
    int getLengthOfText() const;

public slots:
    void setText( const QString &text );

signals:
    void textChanged( const QString& );

private:
    QString m_text;
};

```

Figura 4 – Exemplo de classe com declarações de *signals* e *slots*  
 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 da Figura 4, 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 5:

```
void MyClass::setText( const QString &text )
{
    if( m_text == text )
        return;

    m_text = text;
    emit textChanged( m_text );
}
```

Figura 5 – Implementação de um *slot* com 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 na Figura 5, 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 (2014o), 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 introduzida uma nova sintaxe para realizar essa conexão:

**`connect(sender, &Classe_de_Sender::signal, receiver, &Classe_de_Receiver::slot);`**

Segundo Qt Project (2014o), 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 6 indica um exemplo entre ligações de vários objetos através de *signals* e *slots*, com um exemplo de código-fonte necessário para criar cada conexão:

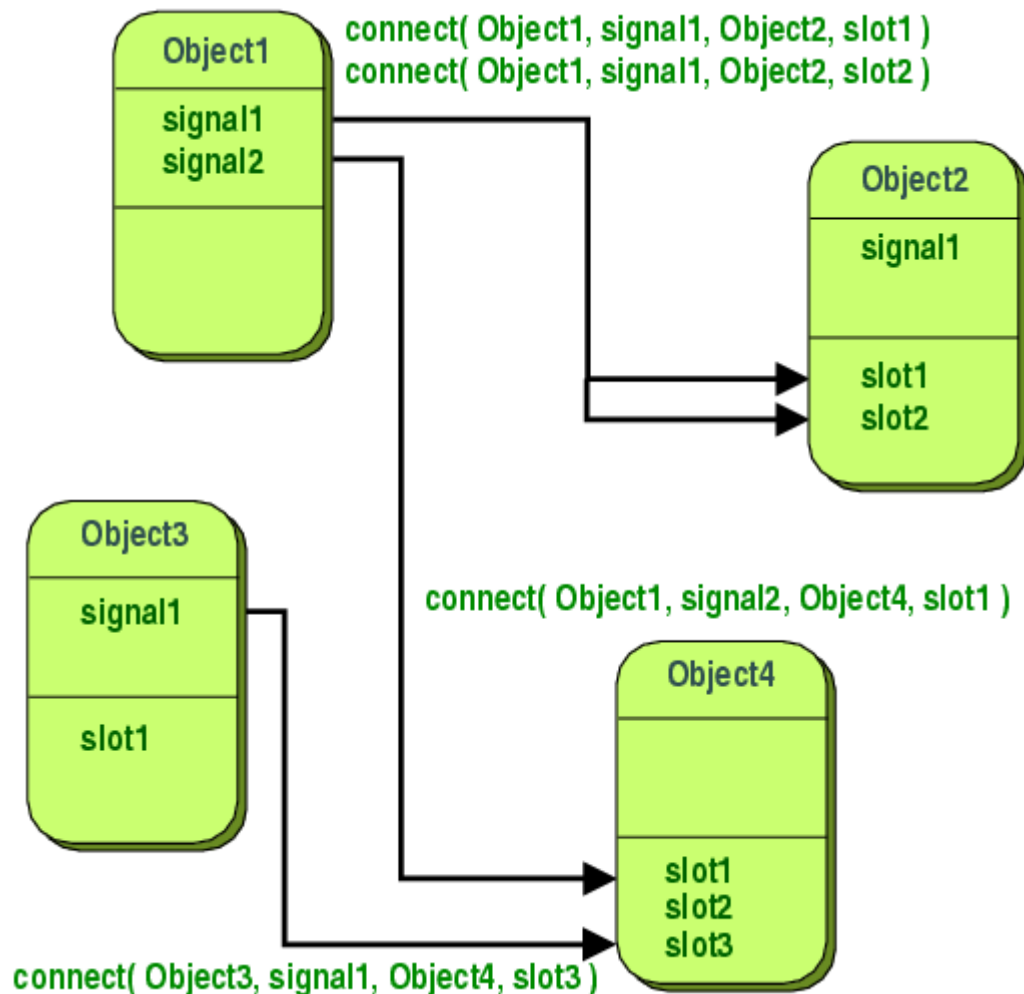


Figura 6 – Exemplo do mecanismo de *signal* e *slot*  
Fonte: Qt Project, 2014n

Um exemplo de utilização deste mecanismo é na utilização de *widgets*<sup>3</sup>. Como exemplificado em Qt Project (2014n) 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

<sup>3</sup> Widgets: “São os componentes primordiais para a criação de UIs no Qt. *Widgets* podem exibir dados e informações de status, receber dados de entrada do usuário, e prover um *container* para outros *widgets* que devem ser agrupados em conjunto.” (QT PROJECT, 2014f)

solução seria realizar uma conexão entre a função de clique do botão e o método `quit()` da tela, como exemplificado na Figura 7:

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QPushButton button("Quit");
    button.show();

    QObject::connect(&button, SIGNAL(clicked()),
                    &a, SLOT(quit()));

    return a.exec();
}
```

Figura 7 – Exemplo de código-fonte com conexão entre objetos utilizando *signals* e *slots*  
Fonte: Molkentin, 2006.

Implementando o código-fonte presente na Figura 7, 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 (2014n), 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.

## 2.3 Qt Creator

Qt Creator, cuja tela inicial é exibida na Figura 8, é uma IDE que provê ao usuário ferramentas para modelar e desenvolver aplicações com o *framework* de aplicação Qt (QT PROJECT, 2014c).

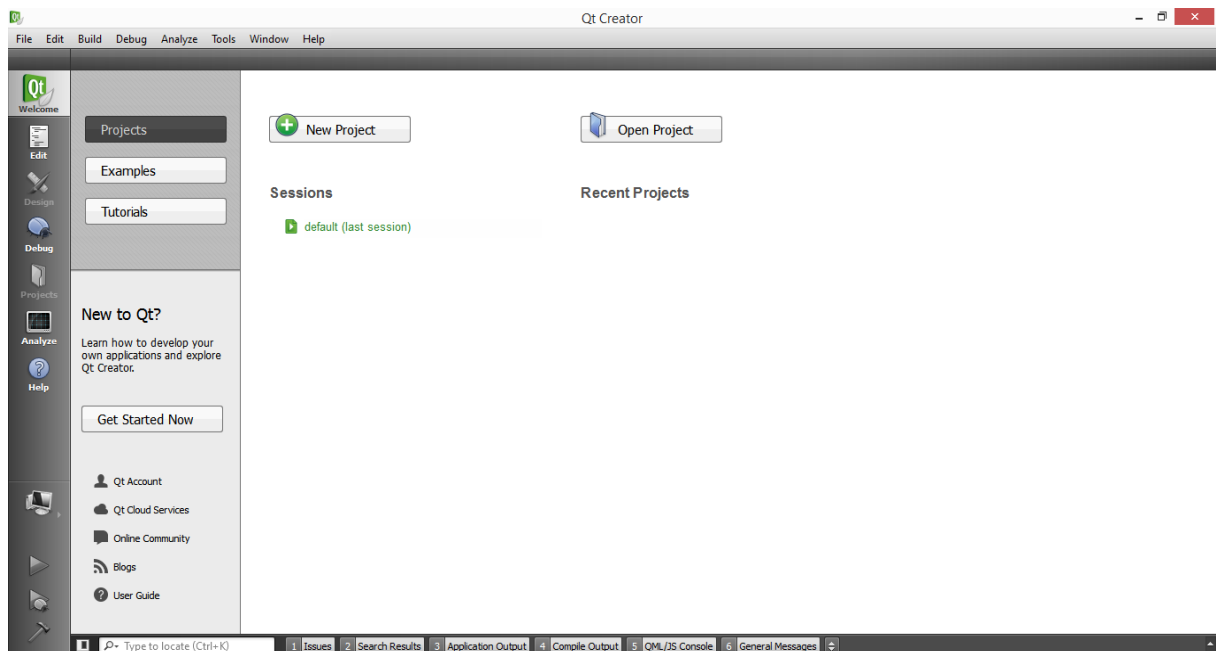


Figura 8 – Tela inicial do Qt Creator  
Fonte: Própria

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, 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 IO, 2014a).

### 2.3.1 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.

#### 2.3.1.1 QML

QML é uma linguagem declarativa que faz parte do *framework* Qt. QML é utilizada no desenvolvimento de aplicativos *cross-platform*<sup>4</sup> 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 9:

---

<sup>4</sup> *Cross-platform*: O mesmo que multiplataforma. A capacidade de uma mesma aplicação poder ser compilada e executada em diversos sistemas operacionais.

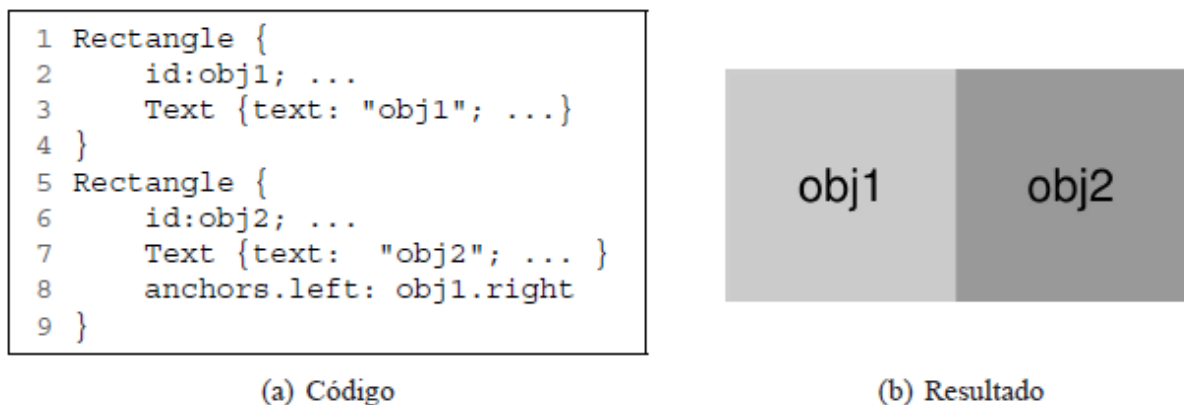


Figura 9 – Exemplo de propriedades atreladas à outras (*property bindings*)

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, como pode ser visto na Figura 10, e que podem ser referências para outros objetos, strings, números, etc., como exemplificado na Figura 11. 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)

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

Figura 10 – Alguns tipos utilizados no sistema de tipagem da linguagem QML.

Fonte: Rosa et al., 2011

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

Figura 11 – 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 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.

### 2.3.2 Qt Widgets

O módulo Qt Widgets está presente no Qt desde as primeiras versões, sendo o módulo mais maduro para a criação de interfaces gráficas. A Figura 12 mostra um exemplo de interface gráfica feita com objetos (*widgets*) do módulo Qt Widgets, e suas respectivas classes:

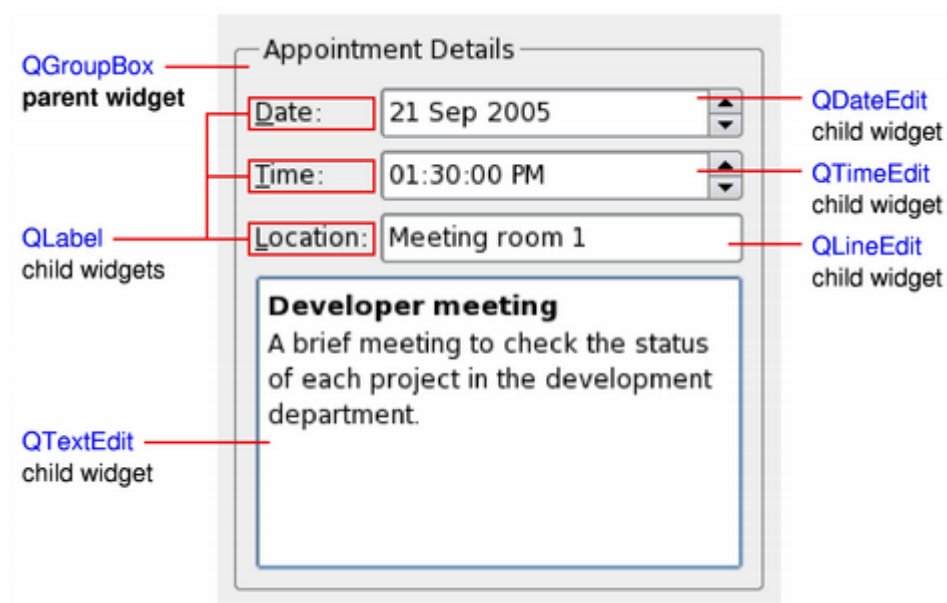


Figura 12 – Exemplo de interface gráfica feita com o módulo Qt Widgets  
Fonte: Qt Project, 2014f

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 13 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).

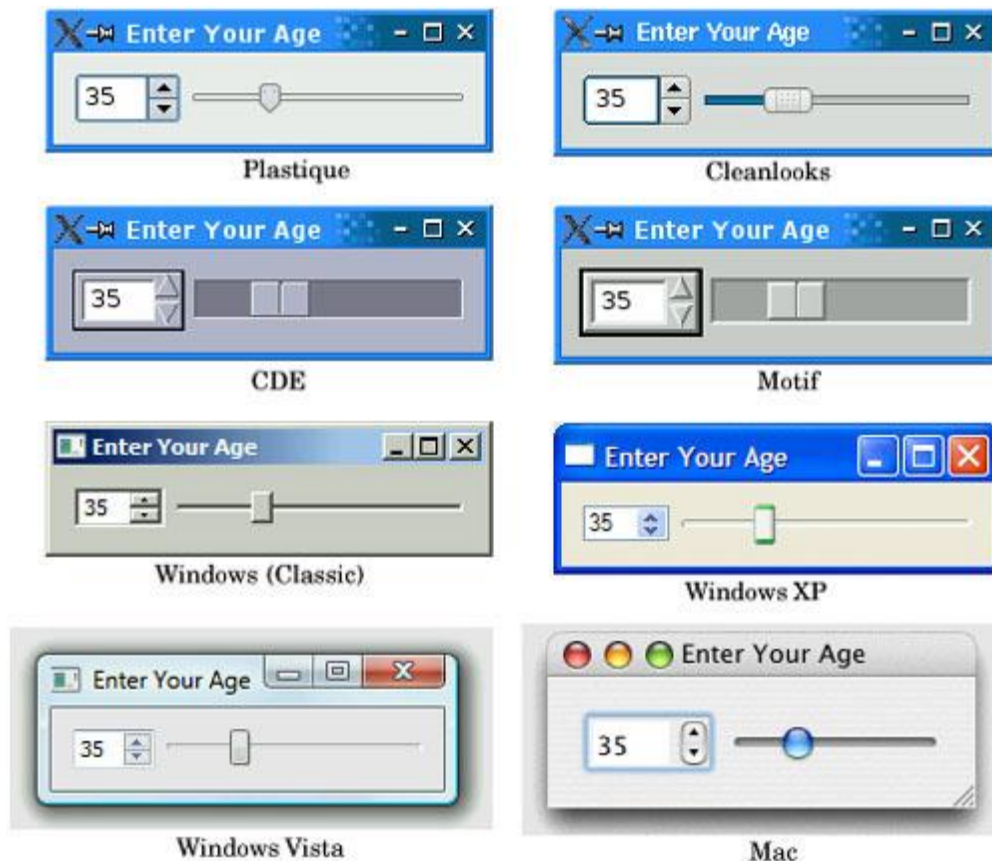


Figura 13 – Exemplo de Estilos dos *Widgets*.  
 Fonte: Blanchette; Summerfield, 2008

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* (como pode ser visto na Figura 14), 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”.

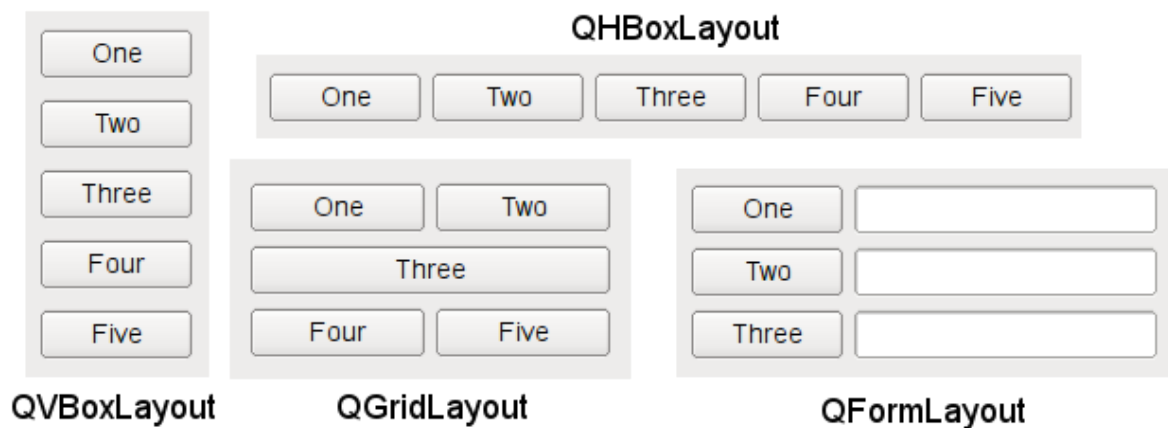


Figura 14 – Exemplos de gerenciadores de *layouts* do módulo Qt Widgets  
Fonte: Qt Project, 2014m

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, 2014m).

## 2.4 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, o KDE.

### 2.4.1 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 15 indica o estado atual da nova interface do Wireshark em funcionamento:

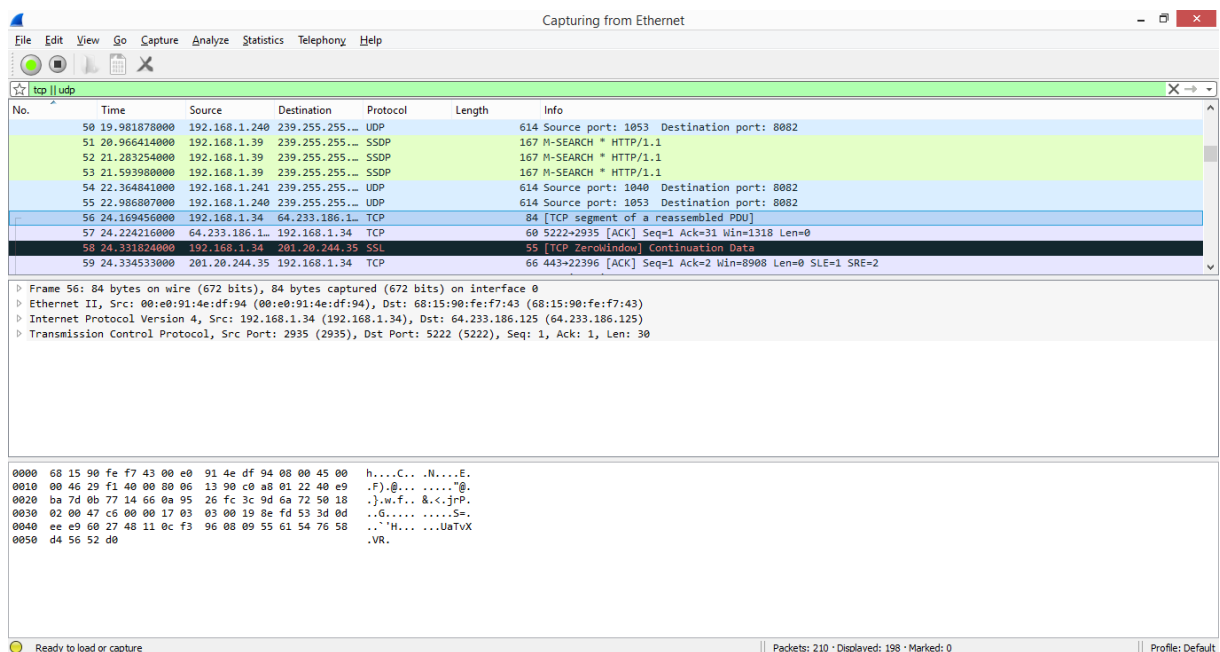


Figura 15 – Wireshark 2 *Preview* em funcionamento  
 Fonte: Própria

### 2.4.2 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 16:

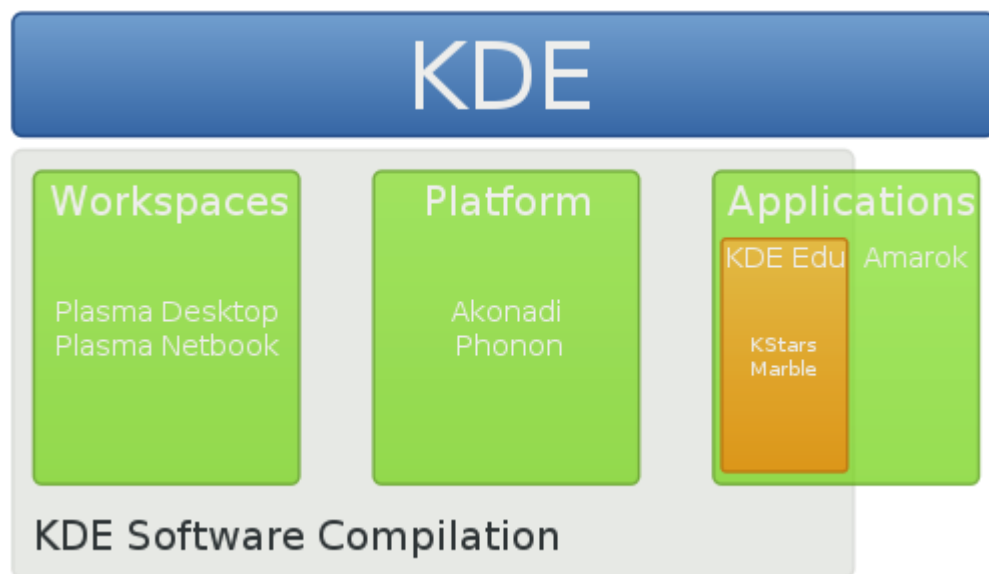


Figura 16 – A comunidade KDE  
Fonte: KDE, 2014c

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 17:

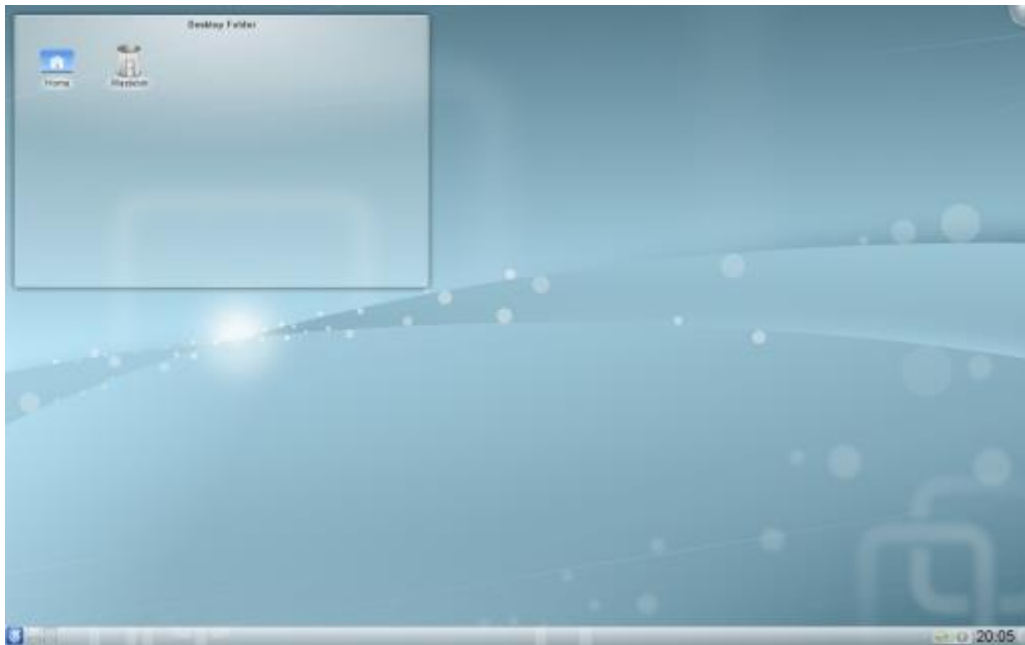


Figura 17 – Interface do Plasma  
Fonte: KDE, 2014b

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 painéis.
- 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*.

## 3 ALTERNATIVAS AO QT

### 3.1 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.

### 3.2 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 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, 2014a) 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 18 apresenta o logo da empresa.



Figura 18 – 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%.

### 3.2.1 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 19 ilustra um trecho de código para a criação de uma interface básica de *login*, e a Figura 20 mostra como essa tela é apresentada por cada um dos sistemas operacionais *mobile*.

```

var profilePage = new ContentPage {
    Title = "Profile",
    Icon = "Profile.png",
    Content = new StackLayout {
        Spacing = 20, Padding = 50,
        VerticalOptions = LayoutOptions.Center,
        Children = {
            new Entry { Placeholder = "Username" },
            new Entry { Placeholder = "Password", IsPassword = true },
            new Button {
                Text = "Login",
                TextColor = Color.White,
                BackgroundColor = Color.FromHex("77D065") }}}
};

var settingsPage = new ContentPage {
    Title = "Settings",
    Icon = "Settings.png",
    (...)
};

var mainPage = new TabbedPage { Children = { profilePage, settingsPage } }

```

Figura 19 – Código para a criação de uma tela de *login* comum  
 Fonte: Xamarin, 2014c

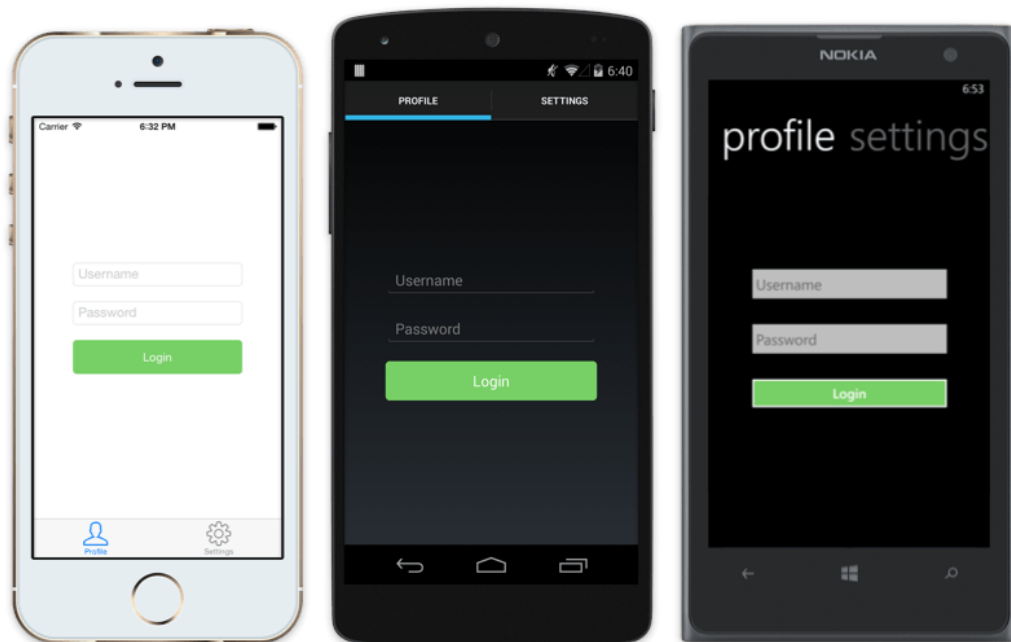


Figura 20 – Código mostrado na Figura 19 rodando, respectivamente, em iOS, Android e Windows Phone  
 Fonte: Xamarin, 2014c

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.

### 3.2.2 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 a 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. 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 21 representa a criação de uma interface gráfica para iOS no Visual Studio da Microsoft, enquanto que a Figura 22 representa a criação da mesma interface no Xamarin Studio.

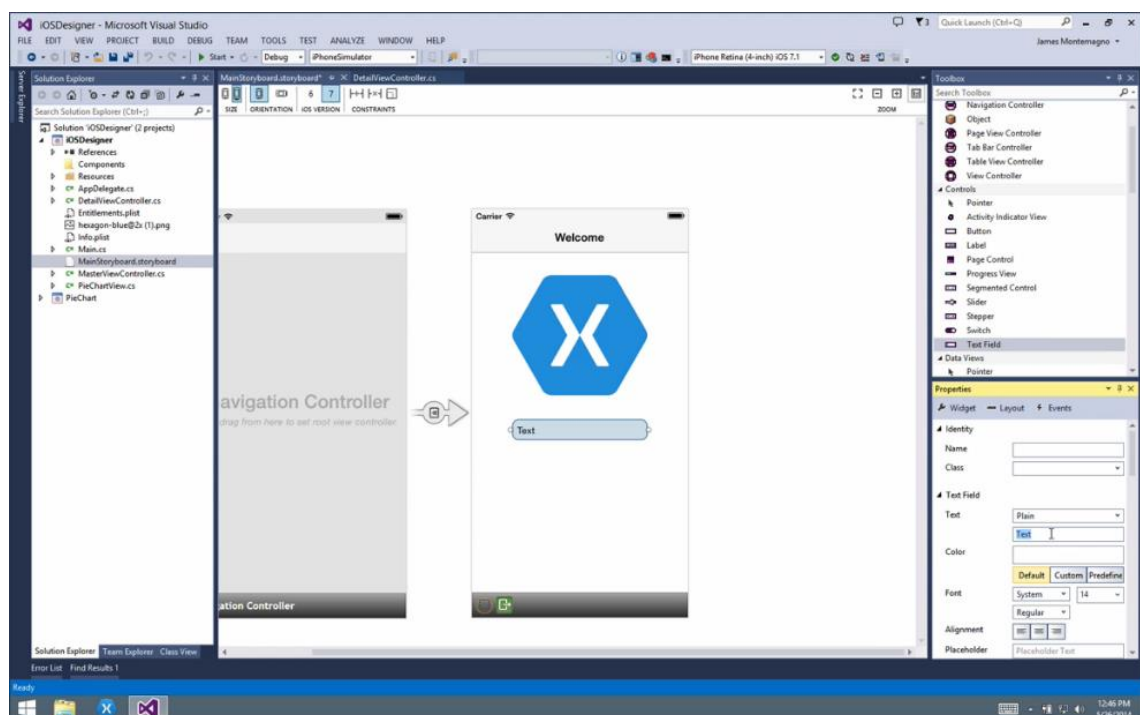


Figura 21 – Criação de interface para iOS utilizando-se o Xamarin para Visual Studio

Fonte: Xamarin, 2014b

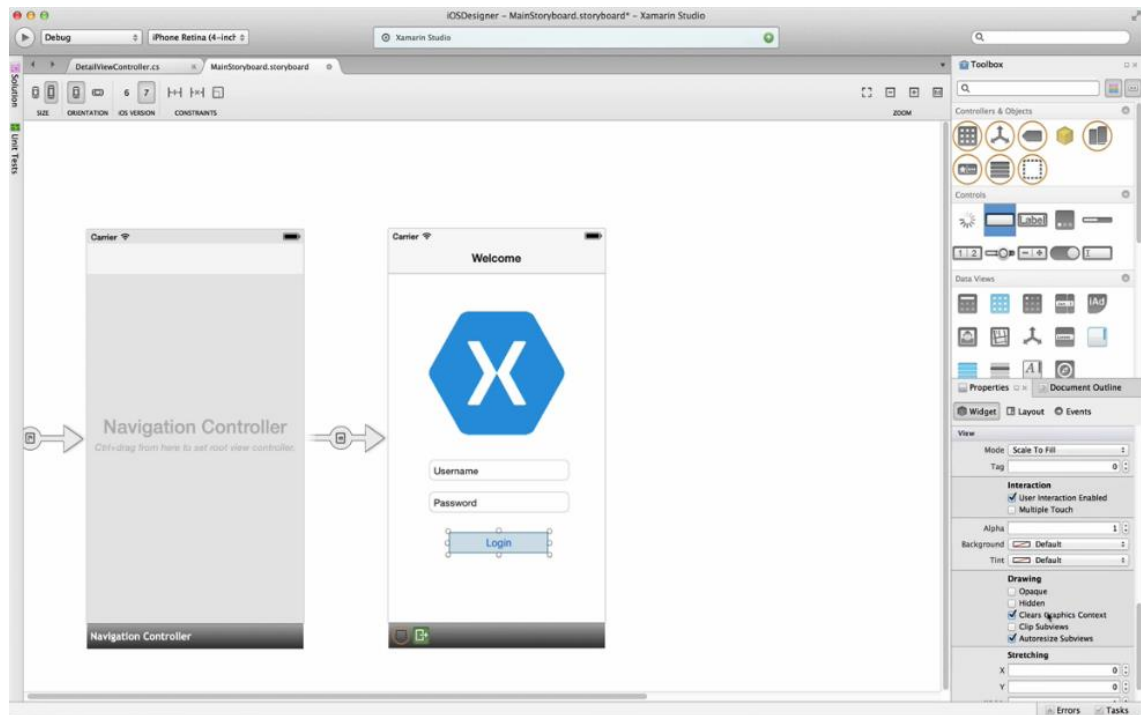


Figura 22 – Criação de interface para iOS utilizando-se o Xamarin Studio  
Fonte: Xamarin, 2014e

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.

### 3.2.3 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 23 representa o Rdio em um iPad rodando iOS e em um *smartphone* rodando Android.



Figura 23 – 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 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 24 mostra a interface do aplicativo desenvolvido com Xamarin que a Kimberly-Clark utiliza na área de vendas de sua empresa.

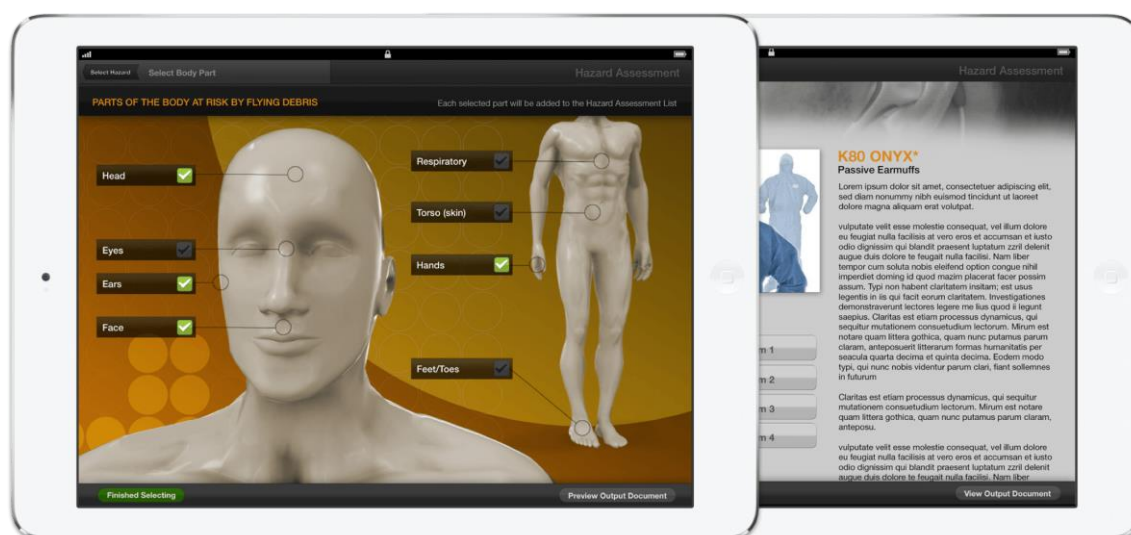


Figura 24 – Aplicativo da empresa Kimberly-Clark desenvolvido com utilização do Xamarin  
Fonte: Xamarin, 2014f

### 3.2.4 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 25 representa a interface do serviço Test-Cloud, para testes automatizados em nuvem.

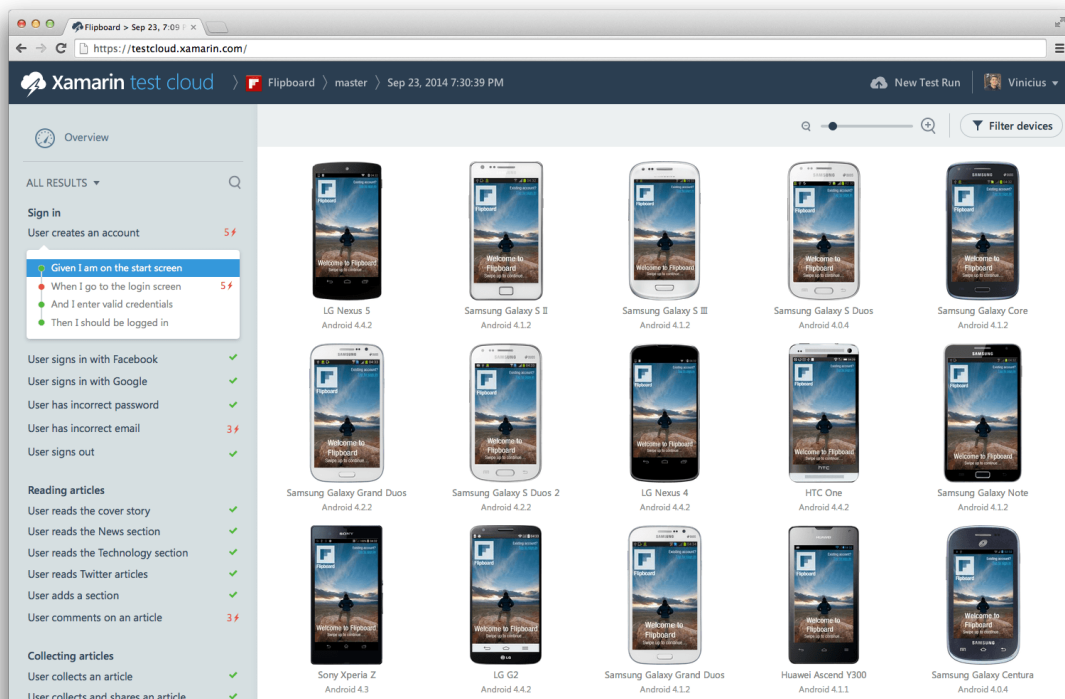


Figura 25 – Interface do Xamarin Test Cloud  
Fonte: Xamarin, 2014h

### 3.2.5 Vantagens e desvantagens

As principais vantagens apresentadas pela plataforma Xamarin são:

- A utilização de uma linguagem única, no caso o C#, facilitando o desenvolvimento; (XAMARIN, 2014b)
- APIs para a criação de interfaces nativas, pois garantem ao usuário maior familiaridade com a interface da aplicação; (XAMARIN, 2014b)
- Possibilidade de utilização do Visual Studio da Microsoft, uma IDE madura e difundida, ao qual o Xamarin pode se integrar através de um *plugin*; (XAMARIN, 2014b)
- Segundo Xamarin, (2014b) aplicativos desenvolvidos na plataforma possuem desempenho nativo, pois são compilados e não interpretados.

Quanto às desvantagens, o principal apontamento é o custo das licenças, que podem parecer elevados se comparados a outras plataformas que oferecem serviços semelhantes gratuitamente. Os valores praticados pela Xamarin atualmente são US\$ 25 mensais para a versão *indie*, US\$999 e US\$1899 anuais para as versões *business* e *enterprise*, respectivamente. (XAMARIN, 2014i).

### 3.3 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 de que oferece suporte a mais de uma linguagem diferente, como por exemplo: 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 26 apresenta o logotipo da Mono:



Figura 26 – Logotipo da Mono  
Fonte: Mono, 2014a

### 3.3.1 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, sendo capaz de compilar aproximadamente 18000 linhas de código por segundo, utilizando para isso uma máquina comum.

A máquina virtual oferecida pela plataforma, a Mono Runtime, possui dois tipos de compiladores: o primeiro 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, partes do código fonte são traduzidos para linguagem de máquina e executados imediatamente. O outro utiliza a tecnologia AOT (*Ahead-of-Time*), que faz uma pré-compilação do código assembly, 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, sendo exemplos:

- Um sistema de coletor de lixo (*garbage collector*), que remove dados desnecessários da memória do computador quando deixam de ser usados pelo *software*;
- Tratamento de exceções;
- Interface com o sistema operacional;
- Gerenciamento de *threads*.

A Mono Runtime ainda conta com a vantagem de poder funcionar em sistemas embarcados (MONO, 2014d).

Ainda como componentes da Mono se encontram uma biblioteca de classes básicas, providas do .NET *framework*, fornecido pela Microsoft, e uma biblioteca de classes da própria Mono, que oferece algumas funcionalidades adicionais. Tais funcionalidades são úteis principalmente para desenvolvimento voltado para a plataforma Linux.

### 3.3.2 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 e nos sistemas operacionais dos consoles Nintendo Wii e Playstation 3. (MONO, 2014e). 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, 2014g).

A Figura 27 apresenta uma tabela retirada do site da própria Mono, que mostra as arquiteturas suportadas e os respectivos sistemas operacionais.

Supported Architectures	Runtime	Operating system
s390, s390x (32 and 64 bits)	JIT	Linux
SPARC (32)	JIT	Solaris, Linux
PowerPC	JIT	Linux, Mac OSX, Wii, PlayStation 3
x86	JIT	Linux, FreeBSD, OpenBSD, NetBSD, Microsoft Windows, Solaris, OS X, Android
x86-64: AMD64 and EM64T (64 bit)	JIT	Linux, FreeBSD, OpenBSD, Solaris, OS X
IA64 Itanium2 (64 bit)	JIT	Linux
ARM: little and big endian	JIT	Linux (both old and new ABI), iPhone, Android
Alpha	JIT	<b>not maintained</b> . Linux
MIPS	JIT	Linux
HPPA	JIT	<b>not maintained</b> Linux

Figura 27 – 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 compilar códigos para funcionar em sistemas operacionais embarcados.

### 3.3.3 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 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 fornece 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 ao 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 dentre uma série de linguagens disponíveis, e permite que ela interopere com códigos escritos em outras linguagens, desde que ambas possuam o suporte à CLR.



### 3.3.4 IDE

A IDE MonoDevelop é um ambiente de desenvolvimento criado para a programação utilizando a plataforma Mono, e foi feito para utilizar exclusivamente linguagens providas 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:

- Suporte a plataformas variadas, sendo elas Windows, Linux e Mac OS X;
- Uma ferramenta semelhante ao IntelliSense da Microsoft, com a função de autocompletar o código;
- Layouts de janelas totalmente customizáveis;
- Suporte a várias linguagens, entre elas C#, Visual Basic.NET, C e C++;
- *Debugger* para realizar o *debug* tanto de aplicações Mono quanto aplicações nativas;
- O módulo GTK# *Virtual Designer* para o desenvolvimento de interfaces;
- Possibilidade de criação de projetos ASP.NET.

A Figura 28 apresenta a janela principal da IDE MonoDevelop, enquanto a Figura 29 apresenta a tela de criação de interfaces.

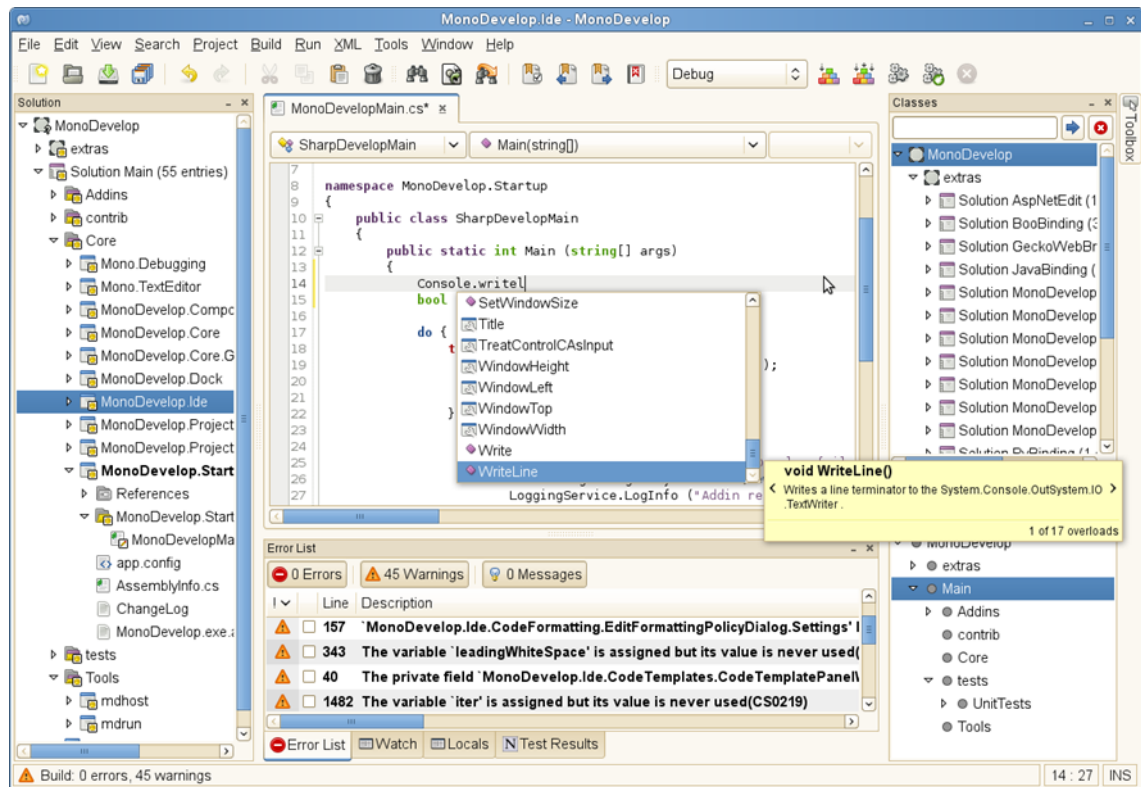


Figura 28 – Tela principal do MonoDevelop  
Fonte: Mono, 2014f

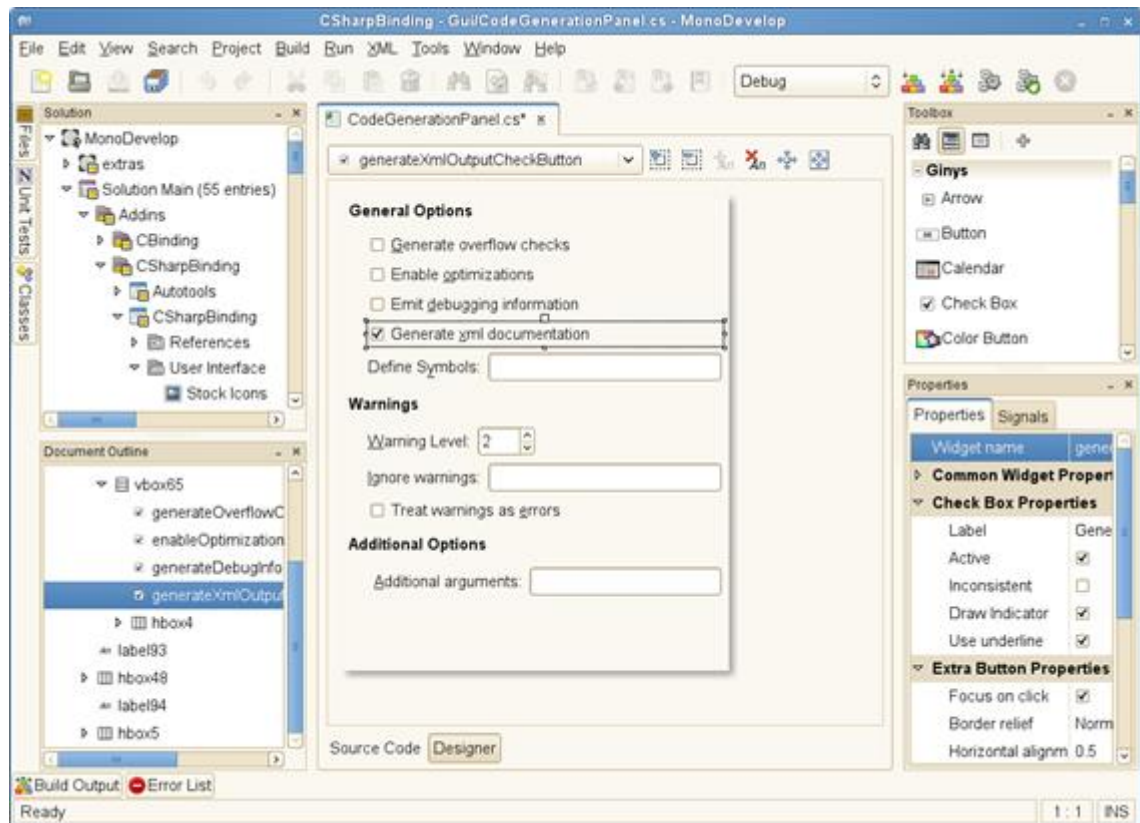


Figura 29 – Tela de criação de interfaces da IDE MonoDevelop  
Fonte: Mono, 2014f

### 3.3.5 Vantagens e desvantagens

As principais vantagens do Mono são:

- Plataforma *open-source*, fazendo com que sua utilização possua custo bem baixo; (MONO, 2014a)
- Alto desempenho, pois combina elementos de compilação AOT com interpretação JIT para otimizar o processo de execução de uma aplicação; (MONO, 2014d).
- Suporte a várias linguagens diferentes, na mesma plataforma. (MONO, 2014a).

A principal desvantagem é que por se basear na plataforma *.NET framework*, o Mono fica, de certa forma, preso às funcionalidades implementadas ao *framework* da Microsoft.

### 3.4 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 uma plataforma. Para tornar isso possível, foi utilizado o conceito de máquina virtual, que será explicado ainda nesse capítulo.

Segundo Deitel e Deitel (2009) 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 30 apresenta o logo do Java:



Figura 30 – Logotipo do Java  
Fonte: Java, 2014

#### 3.4.1 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 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 navegadores, 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 transformar os navegadores *web* 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 31, 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 31 – Logotipo da empresa Oracle  
Fonte: Oracle, 2014

### 3.4.2 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 por 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.

### 3.4.3 Java Virtual 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 específico para uma plataforma, como exemplificado na Figura 32.

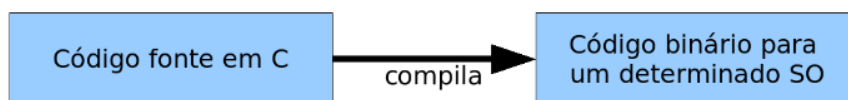


Figura 32 – Processo de compilação em linguagem C  
Fonte: Caelum, 2014a

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 33.

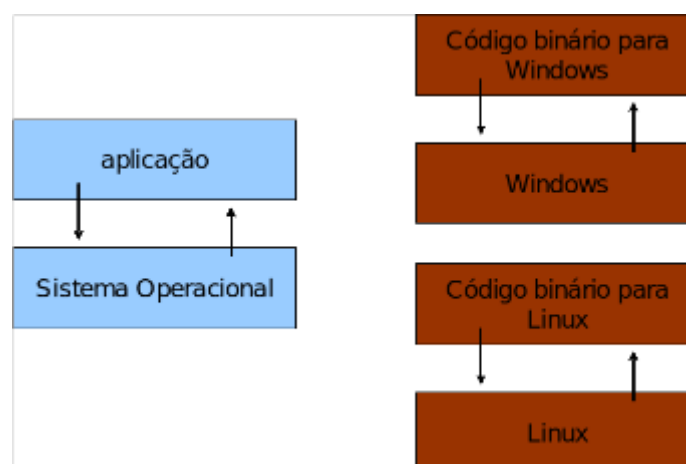


Figura 33 – Comunicação do código binário diretamente com o sistema operacional  
Fonte: Caelum, 2014a

Em Java, esse processo é diferente, pois a linguagem utiliza o conceito de máquina virtual. Segundo Deitel e Deitel (2009) o compilador Java faz a conversão do código Java para *bytecodes*, que são representações das tarefas que devem ser executadas. Os *bytecodes* gerados são executados pela JVM, que se trata da máquina virtual base para a plataforma Java. Máquinas virtuais são *softwares* que simulam computadores, que ocultam seus sistemas operacionais e seu *hardware* das aplicações que interagem com eles. Se uma mesma máquina virtual for implementada para várias plataformas, as aplicações executadas por ela poderão ser utilizadas em todas essas plataformas. A Figura 34 exemplifica o processo de comunicação entre uma aplicação Java e o sistema operacional através da JVM.

Ao contrário da linguagem de máquina, que é dependente do hardware específico do computador, os *bytecodes* são independentes de plataforma – eles não dependem de plataforma de hardware particular. Portanto, os *bytecodes* do Java são portáteis – sem recompilar o código-fonte, os mesmos *bytecodes* podem executar em qualquer plataforma contendo uma JVM que entende a versão do Java em que os *bytecodes* foram compilados. (DEITEL; DEITEL, 2009).

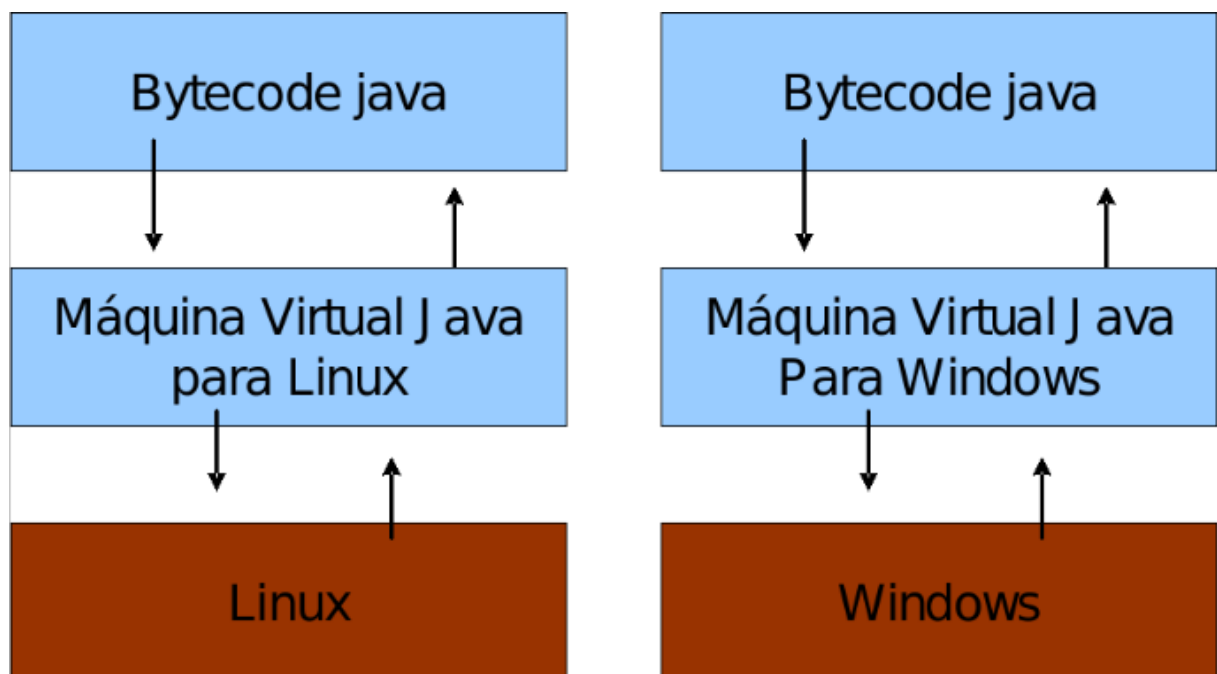


Figura 34 – Processo de comunicação entre aplicação e sistema operacional através da JVM.  
Fonte: Caelum, 2014a

Segundo Deitel e Deitel (2009) as JVMs mais atuais utilizam uma combinação de interpretação e compilação JIT para a execução dos *bytecodes*. Durante esse processo, a JVM busca por *hot spots*, que são partes do *bytecode* executadas mais frequentemente. Nessas partes o compilador Java HotSpot, um compilador JIT, traduz o *bytecode* para linguagem de máquina, e quando a JVM encontra novamente esses trechos compilados, é executado o código de máquina mais rápido.

As Figuras 35 e 36 apresentam um código escrito em Java e o seu *bytecode*, respectivamente.

```

1  class MeuPrograma {
2      public static void main(String[] args) {
3
4          // miolo do programa começa aqui!
5          System.out.println("Minha primeira aplicação Java!!");
6          // fim do miolo do programa
7
8      }
9  }

```

Figura 35 – Exemplo de código escrito em Java  
Fonte: Caelum, 2014a

```

MeuPrograma();
Code:
  0:  aload_0
  1:  invokespecial  #1; //Method java/lang/Object."<init>":()V
  4:  return

public static void main(java.lang.String[]);
Code:
  0:  getstatic  #2; //Field java/lang/System.out:Ljava/io/PrintStream;
  3:  ldc      #3; //String Minha primeira aplicação Java!!
  5:  invokevirtual  #4; //Method java/io/PrintStream.println:
        (Ljava/lang/String;)V
  8:  return
}

```

Figura 36 – *Bytecode* gerado após a compilação do código mostrado na Figura 35  
Fonte: Caelum, 2014a



### 3.4.4 IDEs

As principais IDEs gratuitas para o desenvolvimento em Java são o Eclipse e o Netbeans. Segundo Caelum (2014b), o Eclipse, exemplificado na Figura 37, atualmente é a IDE líder de mercado, e se encontra na versão 4.4. Ele possui seu código livre e é mantido por um consórcio de empresas liderado pela IBM. Embora ele seja mais utilizado para o desenvolvimento em Java, ele oferece suporte para outras linguagens como C, C++ e PHP através de *plugins* que podem ser baixados e incorporados à IDE.

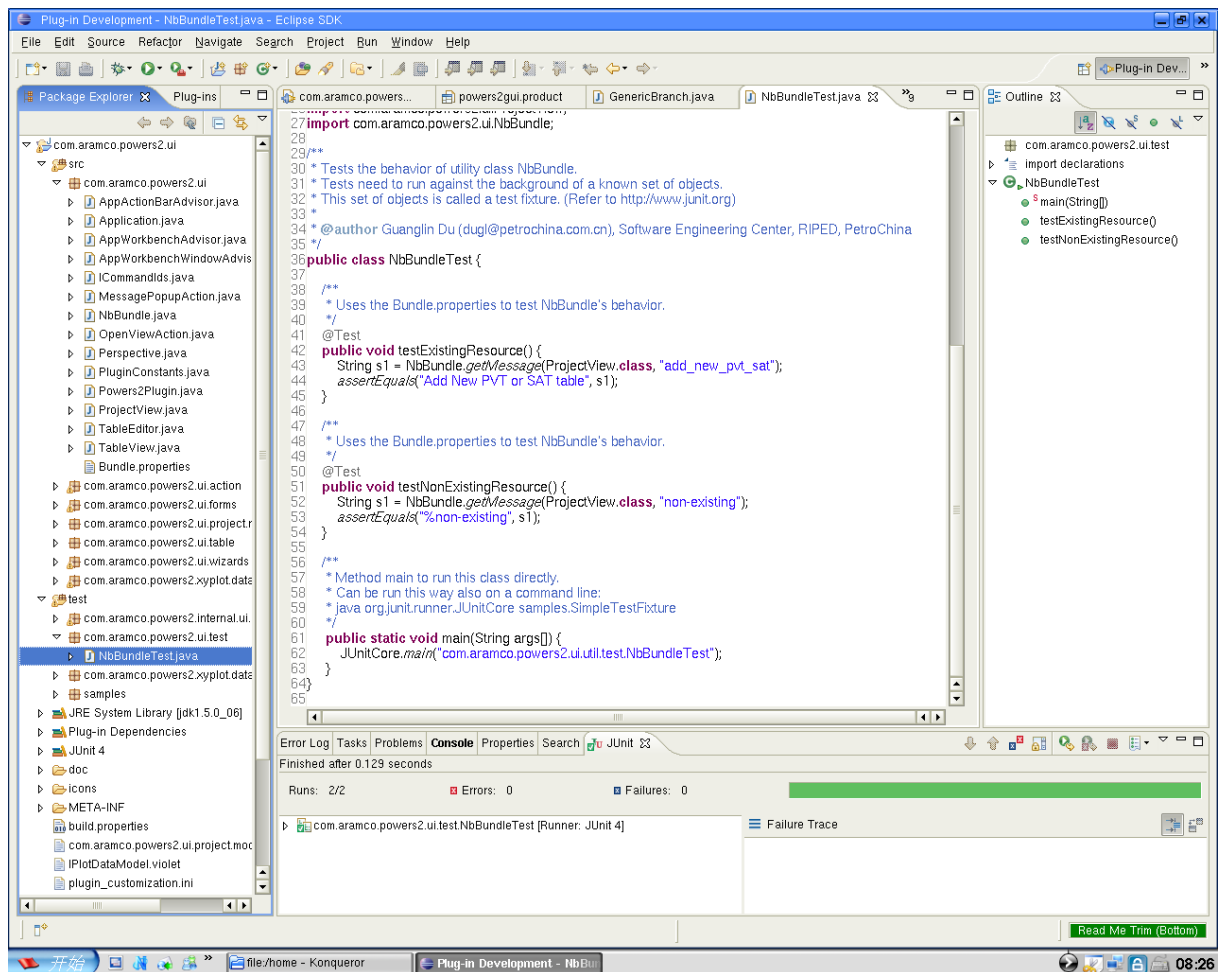


Figura 37 – Ambiente de desenvolvimento da IDE Eclipse  
Fonte: Eclipse, 2014

Já o NetBeans, cuja interface gráfica é exemplificada na Figura 38, começou a ser desenvolvido em 1996 sob o nome de Xelfi. Segundo o site do próprio 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 no ano 2000 tornou seu código público. Atualmente na versão 8.0.1 e mantido pela Oracle, também apresenta suporte para desenvolvimento em outras linguagens como C, C++, PHP e Javascript.

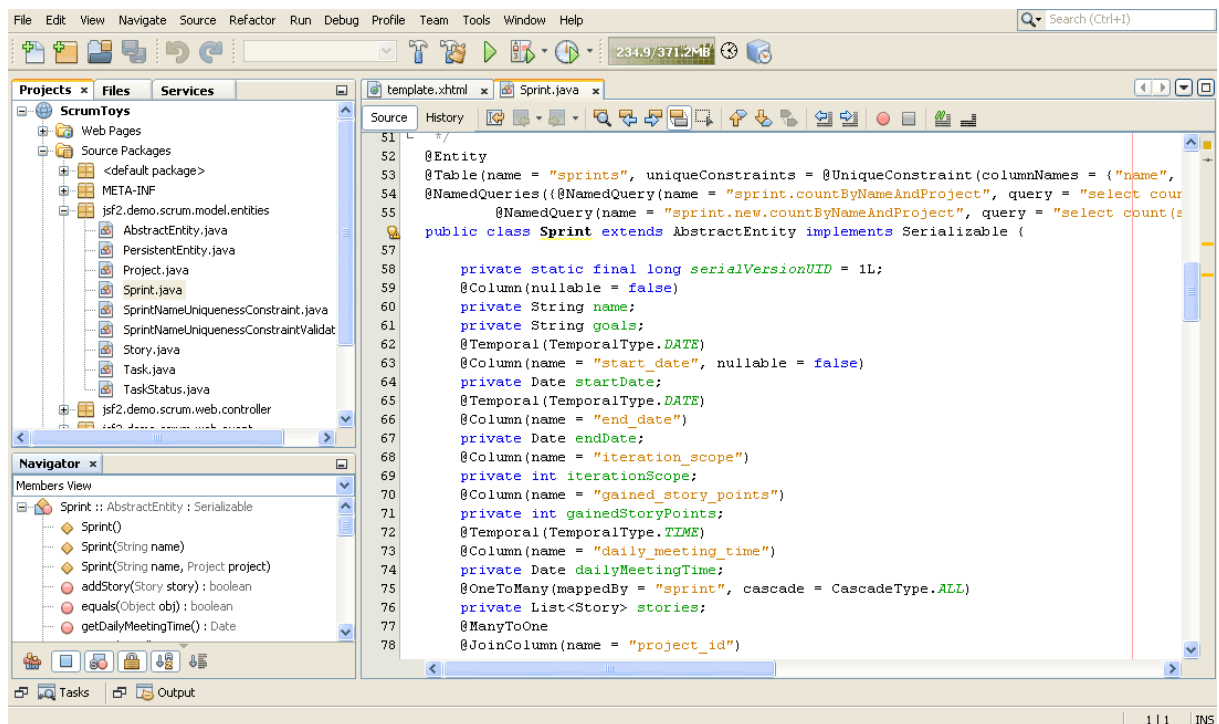


Figura 38 – Ambiente de desenvolvimento do NetBeans  
Fonte: Netbeans, 2014b

### 3.4.5 Interface gráfica em Java

Segundo Horstmann e Cornell (2009) 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, porém a segunda não se trata de uma substituição completa da AWT. O Swing possui componentes GUI mais capazes que o antecessor. Atualmente, usa-se a AWT, especialmente para tratamento de eventos, enquanto a criação de interfaces é feita através da Swing.

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 único, 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 a 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 39 apresenta diferentes *look-and-feels* para uma mesma aplicação.

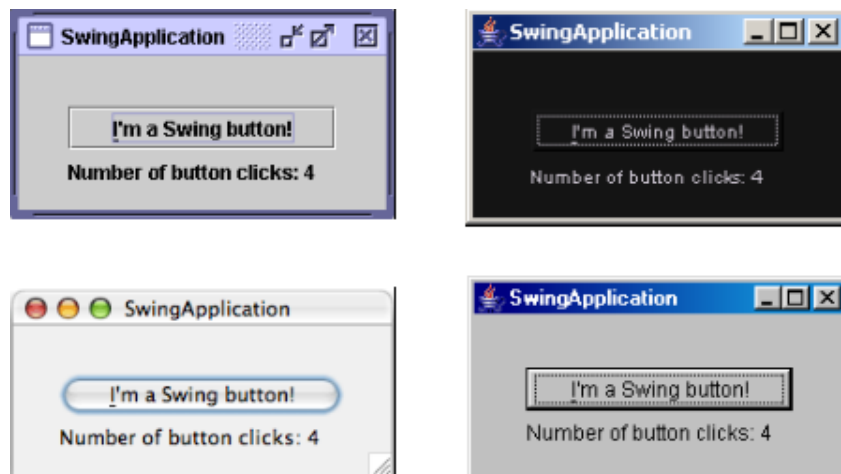


Figura 39 – Utilização de diferentes *look-and-feels* para uma mesma aplicação  
Fonte: Caelum, 2014c

### 3.4.6 Vantagens e desvantagens

As principais vantagens da utilização da plataforma Java são:

- Custo baixo, não sendo necessária a aquisição de nenhum tipo de licença para a utilização da linguagem Java;

- Extenso conjunto de bibliotecas disponíveis gratuitamente a internet (CAELUM, 2014a);
- Segundo Horstmann e Cornell (2009), Java possui a sintaxe mais agradável e semântica mais compreensível que da linguagem C++.

Dentre as desvantagens, a principal é que a plataforma Java, por usar um processo de compilação híbrido, parte compilado, parte interpretado, tende a ter um desempenho inferior a linguagens como o C++ que é totalmente compilado. Embora hoje em dia essa diferença de desempenho seja mínima, ela ainda existe. (DEITEL; DEITEL, 2009).

## 4 ESTUDO DE CASO

### 4.1 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 objetivo do 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 4.4), responsável por fazer a troca de mensagens entre diferentes usuários, independentemente da plataforma, e o servidor (QTCC\_Server, detalhado no tópico 4.5), 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. A Figura 40 representa um modelo de arquitetura da aplicação:

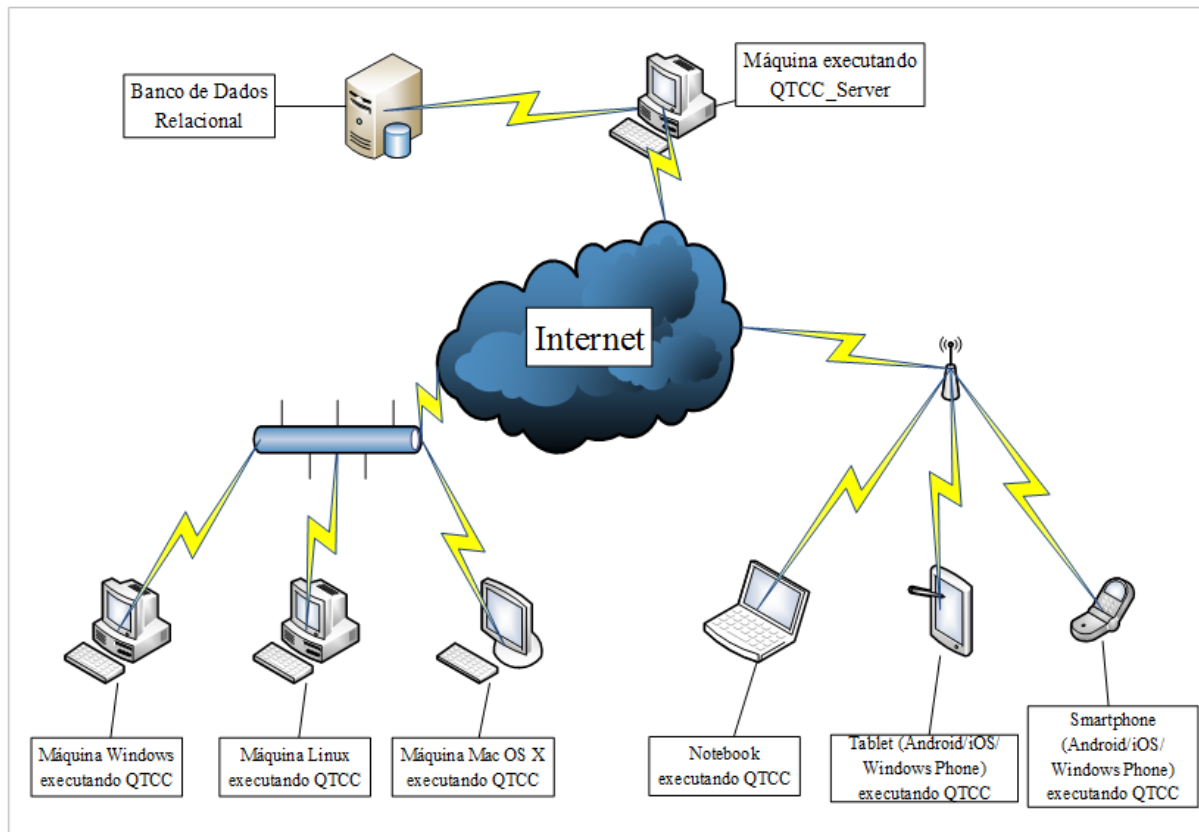


Figura 40 – Modelo de Arquitetura Cliente-Servidor da aplicação  
Fonte: Própria

## 4.2 Tecnologias Utilizadas

### 4.2.1 SQL Server

Para o desenvolvimento do protótipo, foi utilizado o Sistema Gerenciador de Banco de Dados (SGBD) SQL Server, da Microsoft. Esse SGBD se utiliza da linguagem SQL (*Structured Query Language*) para salvar e recuperar dados de seu sistema de banco de dados relacional. A versão utilizada no desenvolvimento do protótipo foi o SQL Server 2012 Enterprise

### 4.2.2 Git

O Git consiste de um sistema de controle de versão gratuito e de código aberto (*open source*). Ele foi utilizado durante todo o desenvolvimento do projeto para salvar todos os

documentos e dados relativos ao protótipo e sua pesquisa, como o documento de monografia, referências bibliográficas digitais e o código-fonte das aplicações cliente e servidor.

As Figuras 41 e 42 mostram a tela principal do programa cliente do Git em modo console (Git Bash) e utilizando interface gráfica (Git GUI), respectivamente:

A screenshot of a Git Bash terminal window. The title bar at the top reads "MINGW32:/c/Users/Ricardo/Desktop/TCC\_FTT" and includes standard Windows window controls (minimize, maximize, close). The terminal content shows the Git welcome message: "Welcome to Git (version 1.9.4-preview20140815)". Below this, it provides instructions: "Run 'git help git' to display the help index." and "Run 'git help <command>' to display help for specific commands.". The prompt "Ricardo@RICARDO ~/Desktop/TCC\_FTT (master)" is displayed in green, followed by a dollar sign "\$" on the next line, indicating the command prompt is ready for input. A vertical scrollbar is visible on the right side of the terminal window.

```
MINGW32:/c/Users/Ricardo/Desktop/TCC_FTT
Welcome to Git (version 1.9.4-preview20140815)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Ricardo@RICARDO ~/Desktop/TCC_FTT (master)
$
```

Figura 41 – Tela inicial do Git Bash  
Fonte: Própria

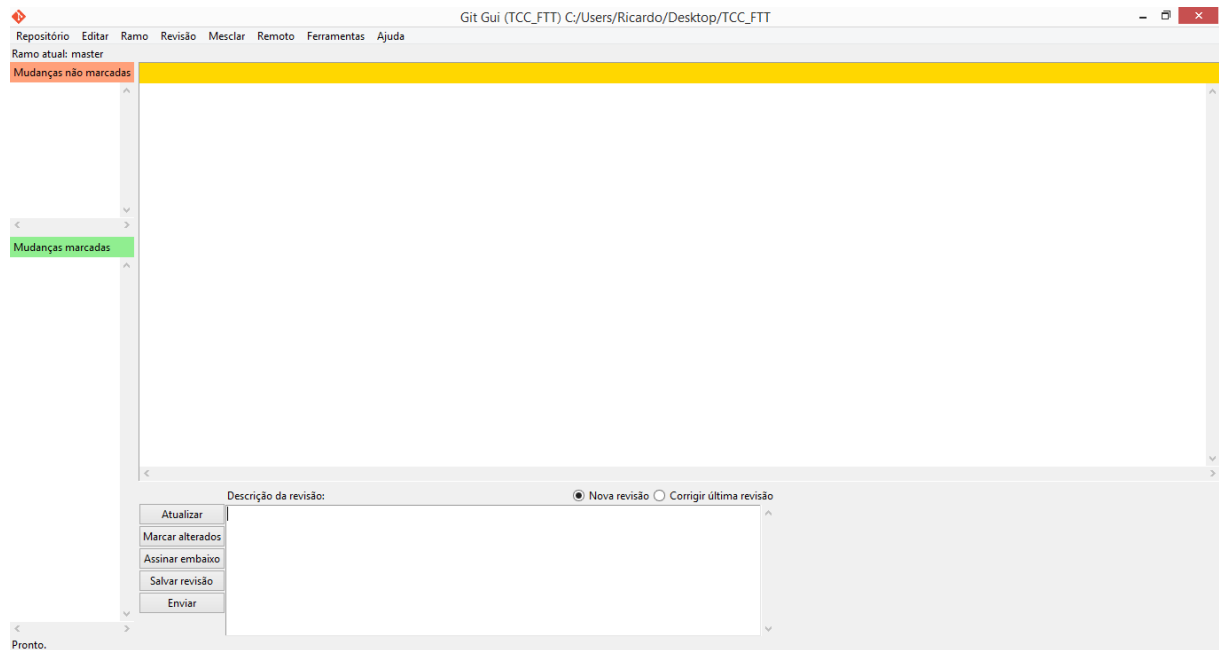


Figura 42 – Tela inicial do Git GUI  
Fonte: Própria

O Git registra o histórico de alterações em cada arquivo por ele controlado, o que o torna essencial para projetos de grandes empresas, onde existe muito controle de erros gerados no código-fonte de uma aplicação.

Outra característica levada em consideração para se utilizar desta ferramenta foi a alta disponibilidade dos arquivos. Um repositório controlado pelo Git é capaz de salvar seus arquivos na nuvem, o que faz com que, ao salvar a alteração realizada em um arquivo, essa alteração seja disponibilizada para todos os usuários que estejam visualizando o projeto, independentemente de onde estejam acessando o repositório. É possível visualizar as alterações registradas no repositório através do próprio Git GUI, o que exibe a tela da Figura 43:



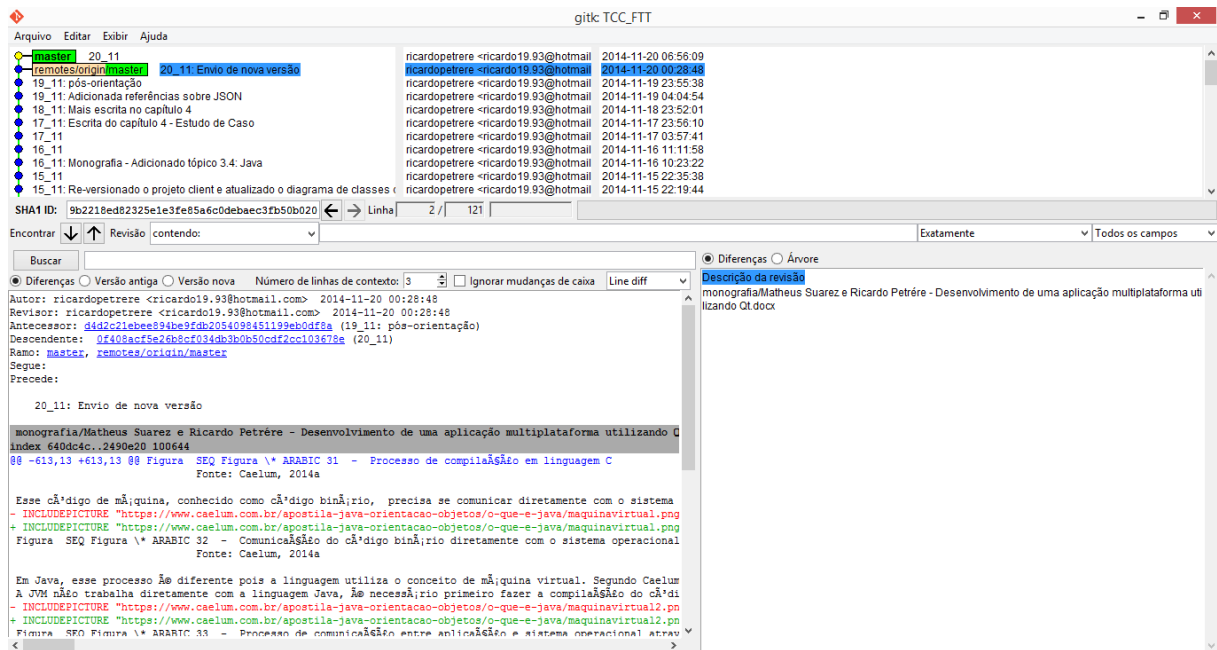


Figura 43 – Visualizando alterações no repositório através do Git  
Fonte: Própria

### 4.3 Delimitações do Projeto

A partir do escopo original do projeto, apenas parte das funcionalidades inicialmente propostas foram implementadas. O protótipo se resume à:

- Realizar cadastro de novos usuários;
- Realizar *login* de usuários;
- Adicionar contatos à um usuário;
- Enviar mensagens de texto entre usuários.

A funcionalidades não implementadas na aplicação incluem:

- Envio de mídias nas mensagens (imagem, áudio e vídeo);
- Cadastro de grupos;
- Conversas em grupo.

O desenvolvimento da aplicação cliente foi realizado nas plataformas Windows e Android, de modo a verificar seu comportamento em um ambiente *desktop* e *mobile*, respectivamente. Os testes e validações foram realizados nas duas plataformas, além da distribuição Linux Ubuntu.

#### 4.4 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 a sua fama mundial, e por ser um dos principais nomes em *frameworks* para desenvolvimento multiplataforma. Durante seu desenvolvimento, foi utilizada a versão 5.3.2 do Qt.

O intuito da aplicação cliente é de realizar *login* de usuários, cadastrar novos usuários, adicionar contatos ao usuário atual, iniciar e gerenciar conversas, e realizar a troca de mensagens entre diferentes usuários.

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).

##### 4.4.1 Interface da aplicação

Como explicado nos tópicos 2.3.1 e 2.3.2, existem dois modos de se desenhar a interface de usuário em uma aplicação utilizando a IDE Qt Creator: Utilizando Qt Quick e utilizando Qt Widgets.

Ao longo do curso, as disciplinas voltadas à programação abordaram o processo de desenhar as UIs no modo tradicional, mais voltado para aplicações *desktop*, com componentes possuindo tamanhos previamente especificados e formatos padronizados, como na Figura 44:

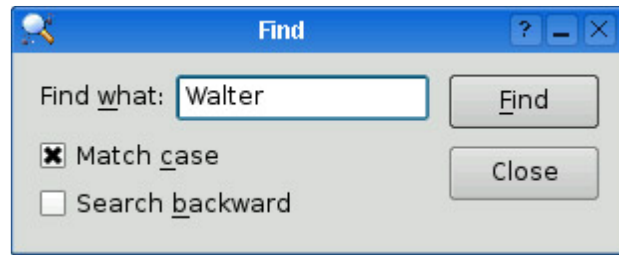


Figura 44 – Exemplo de interface de uma aplicação *desktop*  
Fonte: Blanchette; Summerfield, 2008

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.

O uso de *layout managers* (visto dentro do tópico 2.3.2) 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 45 exemplifica o resultado do uso de *layout managers* para o protótipo da tela de mensagens. Na imagem, foram utilizados os S.Os Android, Windows e Linux, respectivamente:

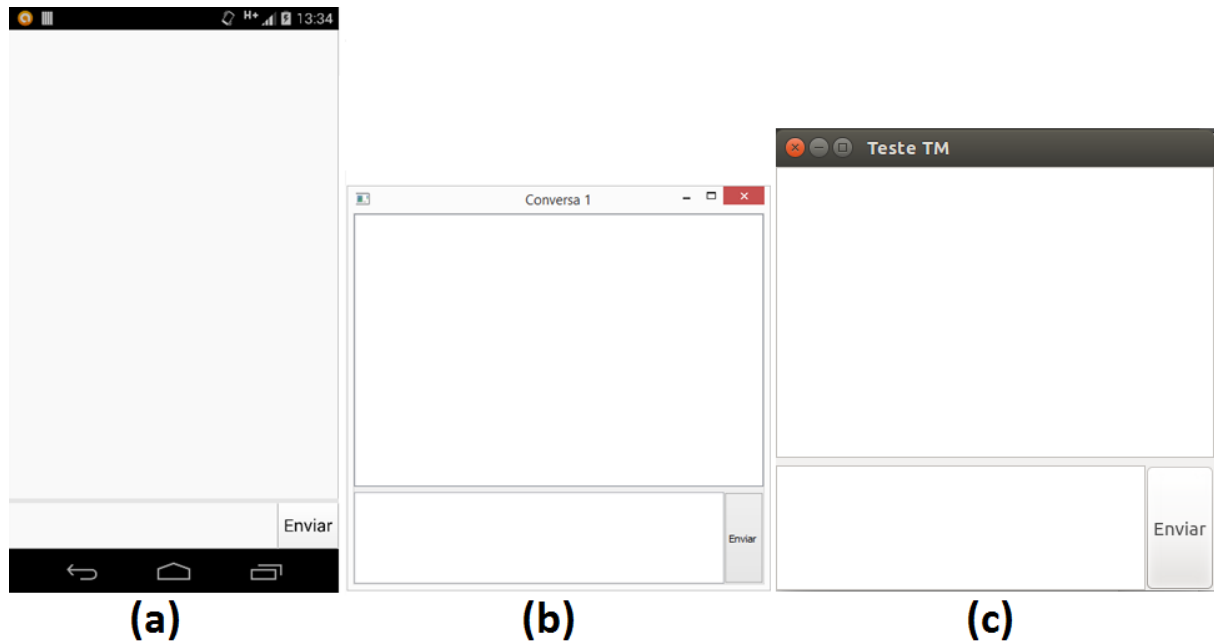


Figura 45 – Protótipo da tela de mensagens na plataforma Android (a), Windows (b) e Linux (c)  
Fonte: Própria

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 foi 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 46. Até então, os autores deste trabalho não encontraram uma solução rápida para o problema, a não ser utilizar janelas normais.

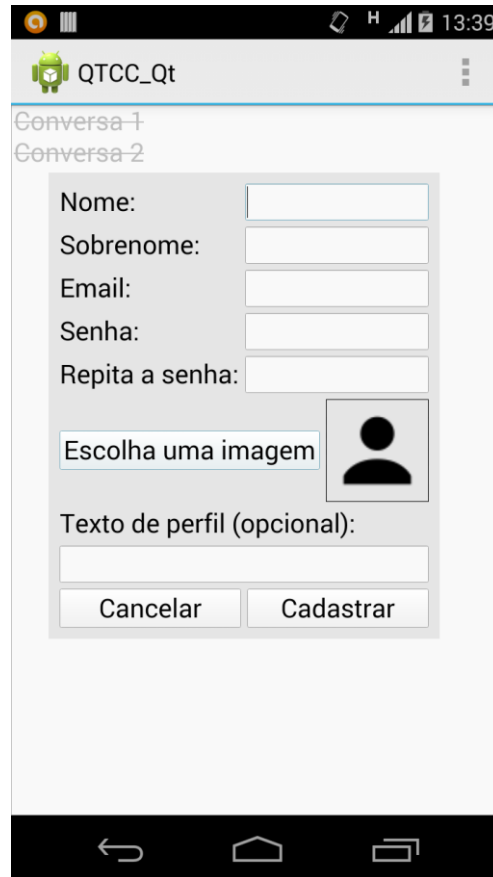


Figura 46 – Exemplo de comportamento ocorrido ao utilizar *dialog* em ambiente Android  
Fonte: Própria

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 47:

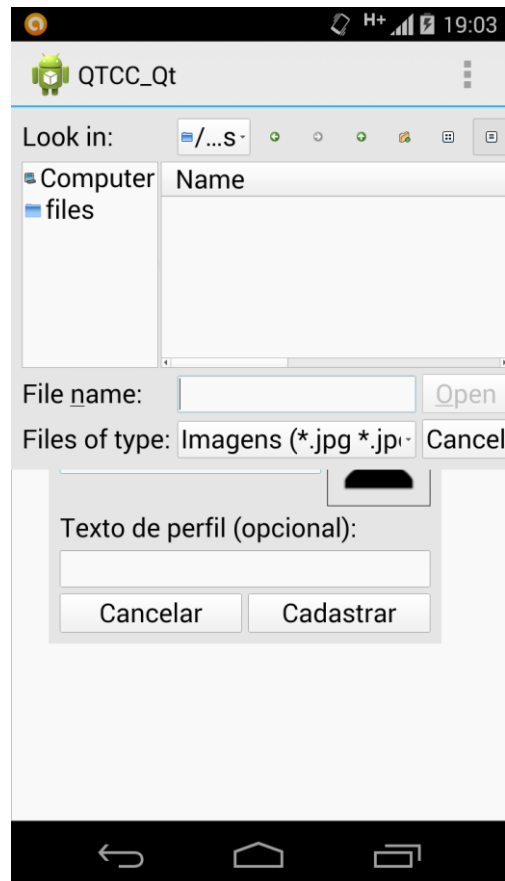


Figura 47 – Exemplo de caixa de diálogo de abertura de arquivo no Android  
Fonte: Própria

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.

#### 4.4.2 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 48:

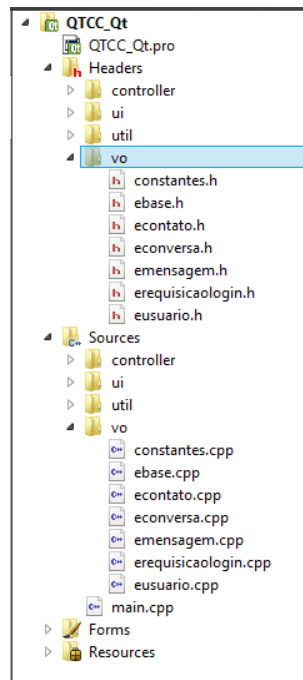


Figura 48 – Classes contidas na pasta "vo"  
Fonte: Própria

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 4.4.3) 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 um dos principais formatos 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 2.3.1.1), como visto na Figura 49:

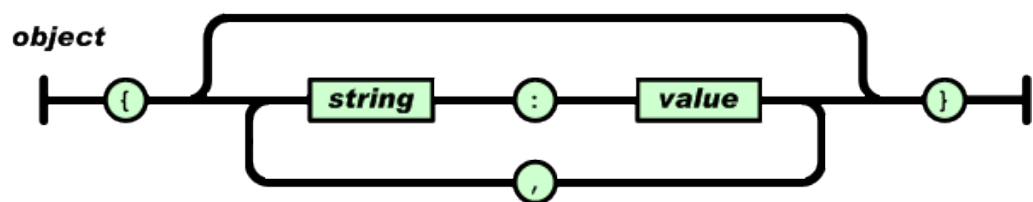


Figura 49 – Propriedade em JSON  
Fonte: JSON, 2014

Onde “string” representa o nome de uma propriedade, e “value” representa seu valor. A vírgula indica que pode haver uma lista de propriedades dentro de um objeto, que é delimitado por “{” e “}”.

- Uma lista ordenada de valores. Um vetor contendo diversos sub-objetos, como visto na Figura 50:

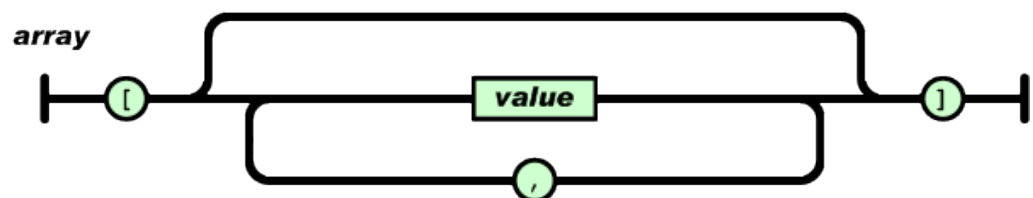


Figura 50 – Vetor em JSON  
Fonte: JSON, 2014

Onde “value” representa um objeto serializado em JSON. A vírgula indica que um vetor, delimitado por “[” e “]”, pode conter inúmeros objetos dentro dele.



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 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.

Um ponto que mereceu destaque durante o uso do JSON foi ao serializar e deserializar vetores de *bytes*. Como pode ser visto no Quadro 1, a variável “Foto” contém a imagem de perfil do usuário ou grupo. Inicialmente se pensou em simplesmente converter a imagem para um vetor de *bytes* e, a partir daí, armazená-la como texto no objeto JSON. Entretanto, foi constatado que a conversão para texto não era concluída, pois, no vetor de *bytes* que representava a imagem, existiam caracteres nulos, e a conversão para texto terminava na primeira ocorrência de tal caractere, o que deixava a imagem incompleta para envio e recebimento.

A solução encontrada foi converter o vetor de *bytes* original, com base em 8 bits por posição, para um vetor com base de 64 *bits* e vice-versa (como se pode perceber nos métodos “array.toBase64” e “QByteArray::fromBase64”). O novo vetor ocupa mais posições do que o primeiro em memória, mas ele pode ser enviado como texto sem perder as informações originais da imagem.

```
//Com base no objeto em JSON, retorna o EContato ao qual se refere
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();
//Deserialização dos bytes do objeto JSON (com codificação Base64) para os bytes da
imagem
QImage a;
QByteArray array = json[Campos::Foto].toString().toLatin1();
//Conversão dos bytes e carregamento da imagem a partir dos bytes convertidos
//É necessário restaurar o vetor de bytes para se ter a imagem original
if(a.loadFromData(QByteArray::fromBase64(array)))
    c._foto = a;
return c;
}

//Com base em um EContato, retorna sua representação em JSON
QJsonObject EContato::Serializar(EContato c)
{
    QJsonObject json;
    json[EBase::Campos::ID] = c._id;
    json[Campos::Nome] = c._nome;
    //Retornar vetor de bytes da imagem
    QByteArray array;
    QBuffer buffer(&array);
    buffer.open(QIODevice::WriteOnly);
    c._foto.save(&buffer,"PNG");
    //Serialização (e conversão) dos bytes da imagem para codificação Base64
    //Essa conversão é necessária para não haver problemas com caracteres nulos na imagem
    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

#### 4.4.3 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`:

```
//Envia o pacote recebido via rede, e retorna o que foi recebido
QString ComunicacaoRede::enviaPacote(QString pacote)
{
    socket = new QTcpSocket(this);
    //Entra em loop até conseguir se conectar com o servidor
    while(!(socket->state()==QAbstractSocket::ConnectedState))
    {
        socket->connectToHost("localhost",5500);
        //Caso não consiga, lançar log de erro
        while(!socket->waitForConnected(10000))
        {
            Logger::debug("não conseguiu");
        }
    }
    QString retorno;
    socket->write(pacote.toLatin1());
    //Espera indefinidamente até que se envie todo o pacote
    socket->waitForBytesWritten(-1);
    //Este método libera o socket para receber a resposta do servidor
    socket->flush();
    while (socket->waitForReadyRead(-1))
    {
        //Incrementar a variável de retorno com o que foi lido até então
        retorno += QString(socket->readAll());
    }
    //Se o pacote indicar que ocorreu erro no envio, retornar nada
    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 no Quadro 3:

```
//Enumerador com os diferentes tipos de pacote de dados enviados ao servidor
enum ETiposPacotesDadosEnum{
    RequisicaoLogin = 1,
    ReceberNovasMensagens=2,
    EnviarNovaMensagem=3,
    StatusContato=4,
    StatusMensagem=5,
```

```

NovoCadastro=6,
EnviarNovoUsuario=7,
EnviarNovoGrupo=8,
BuscaUsuarioPeloEmail=9,
BuscaUsuarioPeloID=10,
BuscaContato=11
};

```

Quadro 3 – Enumerador EtiposPacotesDadosEnum  
Fonte: Própria

## 4.5 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. Isso inclui realizar *login*, cadastrar usuários, enviar mensagens e adicionar contatos.

Por rodar em apenas uma plataforma, não era necessário que o QTCC\_Server fosse desenvolvido no *framework* Qt, portanto foi utilizado o Visual Studio 2013 (e a linguagem de programação C#) para desenvolvê-lo.

O desenvolvimento da aplicação servidor começou no início de outubro e, assim como ocorrido na aplicação cliente, foram criados programas-teste para testar as funcionalidades necessárias (envio de dados pela rede e serialização e deserialização de objetos com JSON)

### 4.5.1 Interface da aplicação

A interface da aplicação servidor é bem simples, sendo de apenas uma tela contendo um botão para reconfigurar o acesso ao banco de dados, a listagem de usuários *online*, além dos contatos do usuário selecionado, como visto na Figura 51:

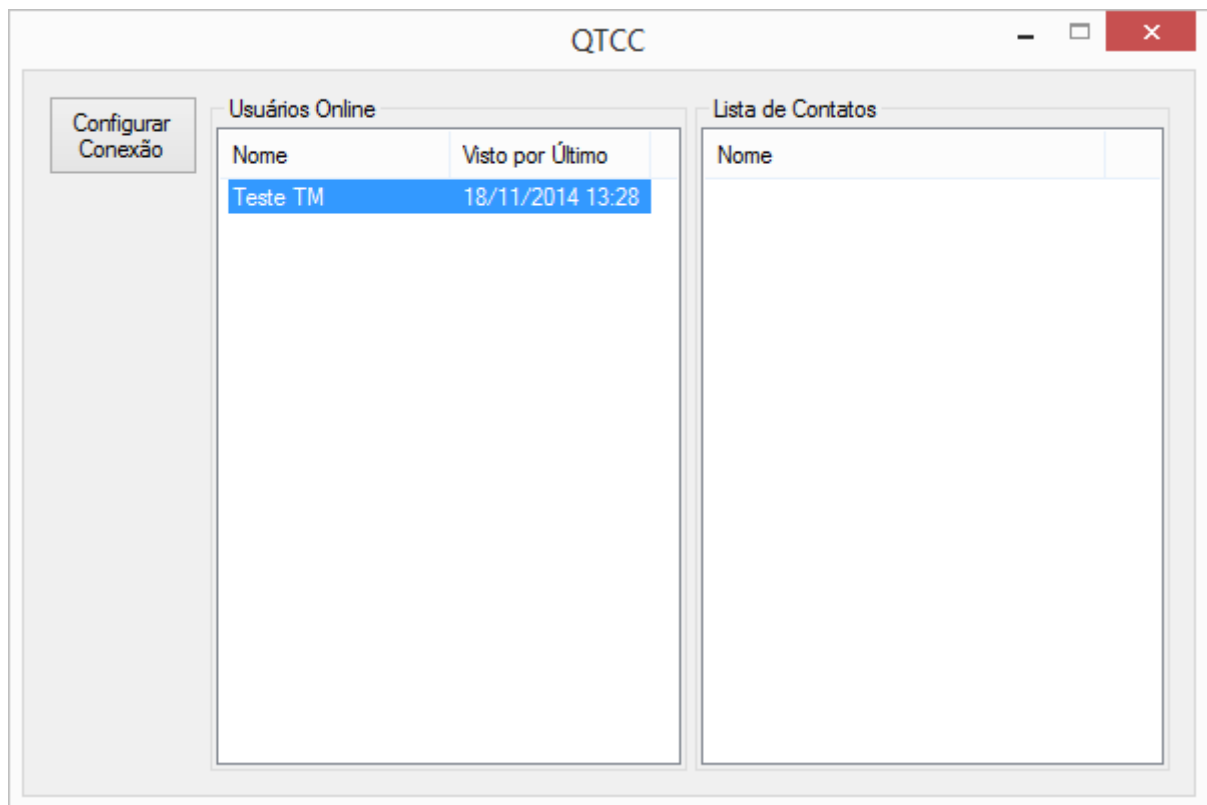


Figura 51 – Tela Principal da aplicação servidor  
Fonte: Própria

A listagem de contatos do usuário só é exibida quando um usuário é selecionado. Caso contrário, o espaço fica em branco, como se pode ver na Figura 52:

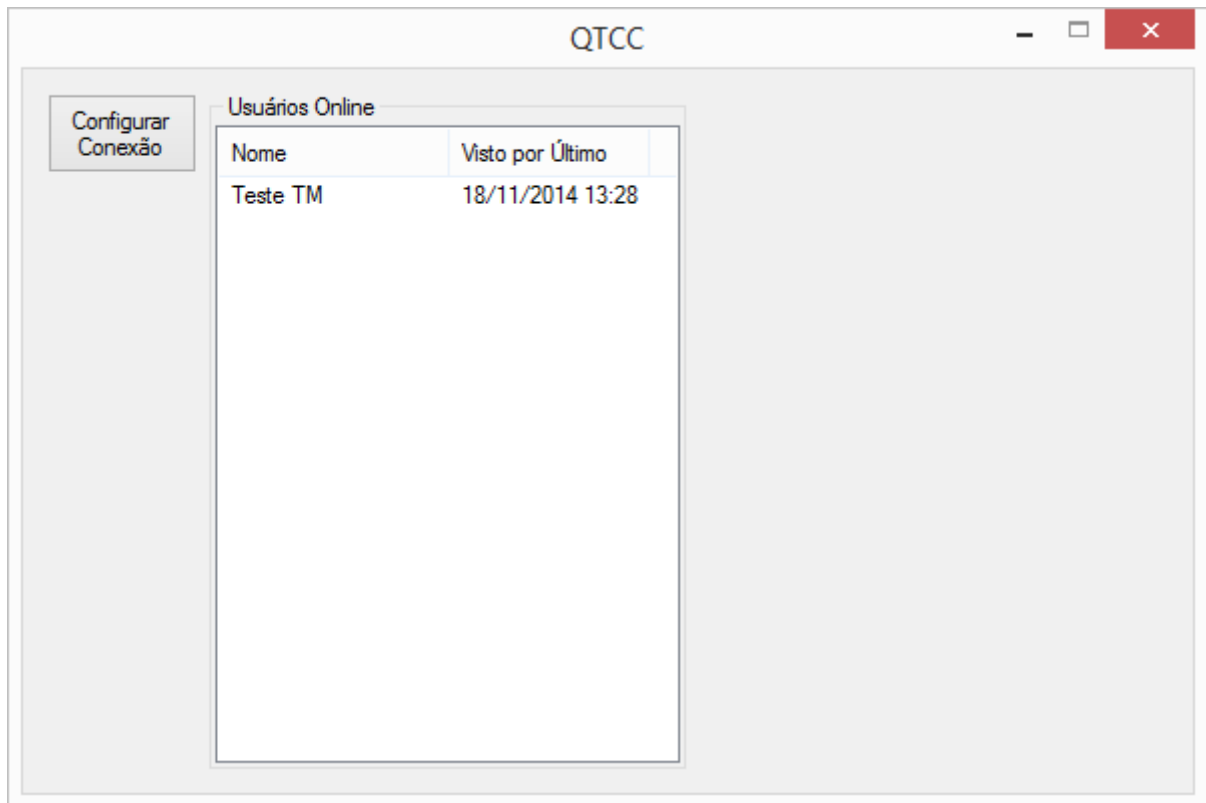


Figura 52 – Tela principal da aplicação servidor sem selecionar um usuário  
Fonte: Própria

Ambas as listagens são atualizadas a cada 10 segundos. Na lista de usuários *online*, o campo “Visto por Último” exibe um valor diferenciado dependendo do tempo em que o usuário se encontra inativo:

- Se o usuário estiver logado há menos de 5 minutos, aparece como “Online”
- Se estiver a menos de uma hora, aparece “Há (quantidade de minutos ausente) minutos”
- Se a última vez online foi no mesmo dia, aparece “Hoje às (hora e minutos)”
- Caso contrário, aparece a data no formato “dia/mês/ano hora:minuto” (dd/MM/yyyy HH:mm).

O botão para configurar o acesso ao banco e sua funcionalidade estão descritos no tópico 4.5.4.

#### 4.5.2 Serialização em JSON

A linguagem de programação C# possui, dentre suas bibliotecas, classes voltadas exclusivamente para a serialização e deserialização de objetos, abrangendo as linguagens XML e JSON. Para tanto, é necessário, em todas as classes que utilizarão a serialização e deserialização, incluir o *namespace* `System.Runtime.Serialization`.

Esse *namespace* não vem adicionado como uma referência padrão dos projetos no Visual Studio, portanto é necessário incluir manualmente a referência. Para isso, foi feito o seguinte:

- Dentro do Visual Studio, clicar com o botão direito do *mouse* no projeto, navegar até “Adicionar” e selecionar “Referência”, como exemplificado (em inglês) na Figura 53:

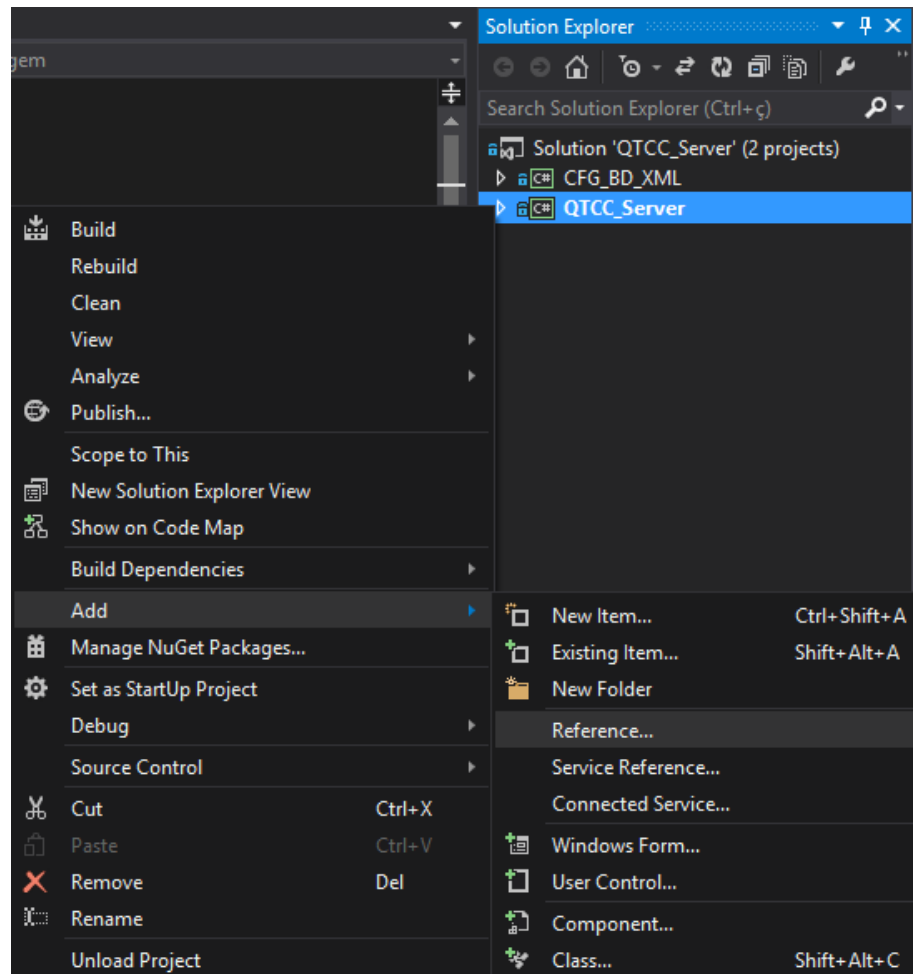


Figura 53 – Selecionando para adicionar uma referência ao projeto em C#  
Fonte: Própria

- Na tela exibida, procurar por System.Runtime.Serialization e marcar a caixa de seleção ao lado, em seguida clicando em “OK”, como se pode ver na Figura 54:



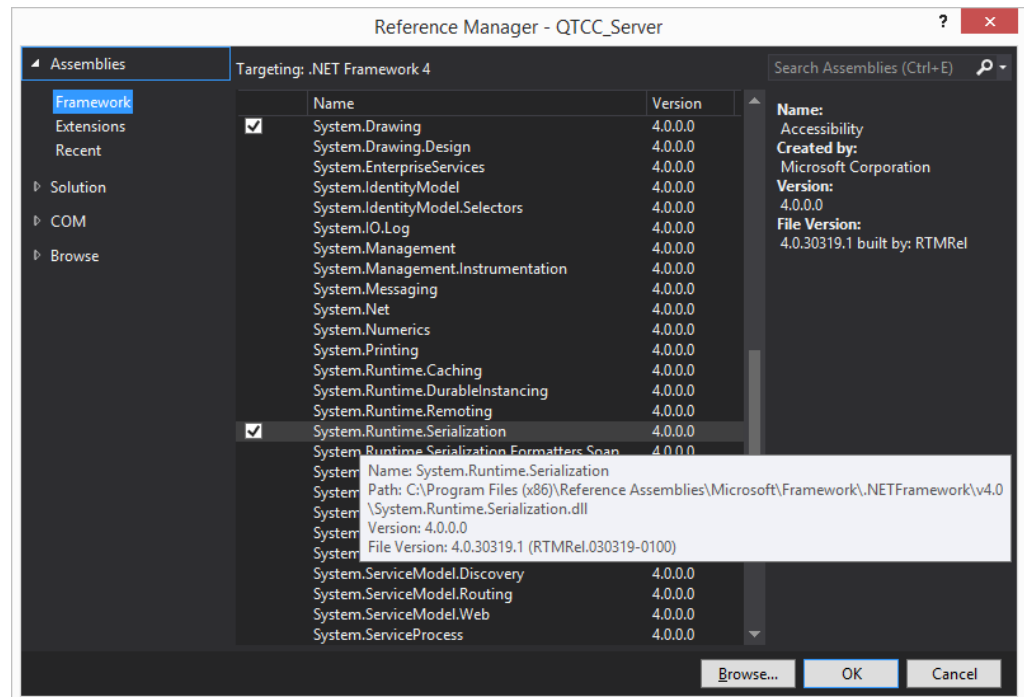


Figura 54 – Adicionando uma referência ao projeto  
Fonte: Própria

Para permitir a serialização e deserialização dos objetos de uma classe, é necessário inserir os seguintes atributos:

- Na linha anterior ao nome da classe, inserir `[DataContract]`. Isso indica que os objetos desta classe podem ser serializados.
- Na linha anterior a cada atributo que se deseja serializar, inserir `[DataMember]`.

Ambos os atributos permitem o uso de parâmetros opcionais em sua declaração. Na aplicação servidor foi utilizado, dentro do atributo `[DataMember]`, o parâmetro “Name”, que define o nome da propriedade quando esta for serializada ou deserializada. Desta forma, foi possível padronizar os nomes das propriedades tanto na aplicação cliente quanto no servidor.

Outra utilidade encontrada no parâmetro “Name” de `DataMember` foi para realizar um intermédio entre o valor serializado e o valor armazenado no objeto, o que se provou útil para propriedades que envolvem *bytes* ou outros objetos.

Na classe `Contato`, existe a propriedade “Foto”, que é composta da imagem de perfil do usuário ou grupo. Como visto no tópico 4.4.2, a imagem de perfil pode conter caracteres nulos dentro de seu vetor de *bytes*, o que impede uma serialização direta. A solução para isso foi criar na classe `Contato` outro atributo, chamado “Foto\_String”, sendo esta a propriedade com o

atributo `DataMember` com o parâmetro “Name” configurado para o nome da propriedade “Foto” no objeto em JSON.

Essa propriedade não armazena nenhum valor, apenas serve para converter os *bytes* da propriedade Foto para texto, que é a forma que a imagem é transmitida no objeto em JSON. O Quadro 4 ilustra o uso e funcionamento das propriedades “Foto” e “Foto\_String”:

```
#region Foto
/// <summary>
/// Representa a imagem de um contato
/// </summary>
public Image Foto
{
    get;
    set;
}
/// <summary>
/// Propriedade que serve para converter os bytes de "Foto" para texto e vice-versa
/// </summary>
// O atributo DataMember Indica que a propriedade abaixo é que será utilizada na
serialização/deserialização
[DataMember(Name = Campos.Foto)]
String Foto_String
{
    get
    {
        //Com "using", é garantido que, no final do bloco de código, o objeto "ms" será
disposto, mesmo ocorrendo exceção
        using (System.IO.MemoryStream ms = new System.IO.MemoryStream())
        {
            //Armazena em "ms" os bytes de "Foto" com o formato de imagem "PNG"
            Foto.Save(ms, System.Drawing.Imaging.ImageFormat.Png);
            //Retorna os bytes de ms em texto com codificação Base64
            return Convert.ToBase64String(ms.ToArray());
        }
    }
    set
    {
        using (System.IO.MemoryStream ms = new System.IO.MemoryStream())
        {
            //Extraí os bytes a partir do texto com codificação Base64
            byte[] a = Convert.FromBase64String(value);
            //Armazena os bytes de "a" em "ms"
            ms.Write(a, 0, a.Length);
            //Gera a imagem a partir dos bytes de "ms", e atribui à "Foto"
            this.Foto = Image.FromStream(ms);
        }
    }
}
#endregion Foto
```

Quadro 4 – Propriedade Foto e Foto\_String  
Fonte: Própria

Para realizar a serialização e deserialização dos objetos da aplicação, foi criada a classe `JSON_Logic`, que possui os métodos estáticos `Deserializa` e `Serializa`.

O método `Serializa` recebe, além do objeto a ser serializado, um identificador do tipo de objeto que será serializado, e retorna o objeto em JSON. O método `Deserializa` recebe o objeto

em JSON e o identificador do tipo de objeto a ser gerado, e retorna o objeto deserializado. A classe JSON\_Logic está presente no Quadro 5:

```

/// <summary>
/// Serializa um objeto qualquer para JSON
/// </summary>
/// <typeparam name="T">O tipo do objeto a ser serializado</typeparam>
/// <param name="item">O objeto a ser serializado</param>
/// <returns>A representação do objeto em JSON</returns>
public static String Serializa<T>(T item)
{
    String retorno;
    //Define o serializador para o tipo do objeto passado
    DataContractJsonSerializer serializer = new DataContractJsonSerializer(item.GetType());
    MemoryStream s = new MemoryStream();
    try
    {
        //Serializa o objeto em JSON, e envia seus bytes para "s"
        serializer.WriteObject(s, item);
        //Retorna os bytes em String
        retorno = Encoding.UTF8.GetString(s.ToArray());
    }
    finally
    {
        //Fecha o buffer de "s"
        s.Close();
    }
    return retorno;
}
/// <summary>
/// Deserializa um JSON para qualquer objeto
/// </summary>
/// <typeparam name="T">O tipo do objeto a ser gerado</typeparam>
/// <param name="dados">O objeto em JSON</param>
/// <returns>O objeto gerado</returns>
public static T Deserializa<T>(String dados)
{
    T retorno;
    //Define o serializador para o tipo do objeto a retornar
    DataContractJsonSerializer serializer = new DataContractJsonSerializer(typeof(T));
    //Instancia "s" já preenchendo-o com os bytes do JSON passado
    MemoryStream s = new MemoryStream(Encoding.UTF8.GetBytes(dados));
    try
    {
        //Lê os bytes contidos em "s" e retorna o objeto do tipo "T"
        retorno = (T)serializer.ReadObject(s);
    }
    finally
    {
        //Fecha o buffer de "s"
        s.Close();
    }
    return retorno;
}

```

Quadro 5 – Classe JSON\_Logic  
Fonte: Própria

### 4.5.3 Troca de dados pela rede

A linguagem de programação C# possui classes e bibliotecas específicas para a comunicação por rede. Para se utilizar destas, é necessário incluir o *namespace* System.Net e System.Net.Sockets em cada classe que utilizará o tráfego de dados via rede.

Na aplicação servidor, foi criada a classe estática denominada ComunicacaoRede, que é responsável por ficar escutando uma porta do servidor em busca de novas conexões vindas da aplicação cliente. Essa classe possui o atributo estático “listener”, da classe TcpListener, que recebe conexões pelo protocolo de rede TCP/IP.

Como o servidor pode receber diversas conexões vindas de inúmeras instâncias da aplicação cliente QTCC simultaneamente, era necessário que, enquanto se estivesse processando uma requisição, a aplicação não ficasse impedida de aceitar novas conexões. Portanto, foi necessário implementar *threads* na classe ComunicacaoRede.

Com base em Thelin (2007), o conceito de *threads* (e, mais amplamente, de processamento paralelo) é de que, em uma mesma aplicação, se possa executar simultaneamente diversos trechos de código, de modo que a aplicação não entre em estado de congelamento enquanto realiza algum processamento mais demorado. Ao delegar um bloco de código (usualmente um método de uma classe) para uma *thread* separada, o programa (e sua *thread* principal) pode continuar seu processamento normalmente, enquanto a outra tarefa é executada em paralelo. Isto é particularmente útil em aplicações que possuem interface gráfica e comunicação simultânea com diversas aplicações, o que se encaixa na aplicação servidor.

Ao se iniciar o servidor TCP, é iniciada a *thread* tEscutaClientes, responsável por se conectar à porta configurada na aplicação em busca de uma nova conexão com uma instância da aplicação cliente. Ao se conectar ao cliente, é iniciada outra *thread*, tReceberPacote, que trata de receber o pacote enviado pelo cliente e responder da forma adequada. A primeira *thread* só possui uma instância em toda a aplicação, enquanto que novas instâncias da segunda são criadas para cada nova conexão estabelecida.

Um comportamento detectado no uso de *threads* foi de que, inicialmente, elas não estavam sendo interrompidas ao fechar a aplicação. Isso se devia ao fato de elas não possuírem nenhuma dependência com a *thread* principal do servidor. A solução encontrada foi, após instanciar cada *thread*, especificar que se tratava de uma *thread* de segundo plano em relação à

execução da aplicação, o que foi feito ao assinalar “true” para a propriedade “IsBackground” da nova *thread*. O Quadro 6 mostra a criação da *thread* responsável por escutar a porta TCP da aplicação:

```

/// <summary>
/// Método responsável por iniciar a escuta da porta TCP
/// </summary>
public static void IniciarServidor()
{
    //Inicializa a Thread que irá escutar a porta TCP, com o método responsável
    Thread tEscutaClientes = new Thread(new ThreadStart(EscutaClientes));
    //Necessário para que a Thread seja eliminada junto com a aplicação
    tEscutaClientes.IsBackground = true;
    //Inicia a execução da Thread
    tEscutaClientes.Start();
}

```

Quadro 6 – Criação da *thread* tEscutaClientes  
Fonte: Própria

Após receber por completo o pacote enviado pelo cliente, é executado o método TrataPacote da classe estática ComunicacaoController. Esse método possui uma validação do cabeçalho do pacote recebido (OP Code) que, dependendo do valor, direciona o tratamento do pacote para um específico método dentro das classes do pacote “Controller”, como se pode ver no Quadro 7:

```

/// <summary>
/// Trata o pacote recebido via rede, redirecionando-o para o método correspondente
/// </summary>
/// <param name="pacote_recebido">O pacote recebido, no esquema
/// (Identificador|dados)</param>
/// <returns>A resposta do servidor</returns>
public static String TrataPacote(string pacote_recebido)
{
    String retorno = "";
    //Separa o pacote, entre identificador (OP Code) e dados
    string[] dados_pacote = pacote_recebido.ToString().Split('|');
    //Tenta converter a primeira parte do pacote para o enumerador de identificadores
    CONSTANTES.TiposPacotesDadosEnum TipoPacote;
    if (Enum.TryParse(dados_pacote[0], out TipoPacote))
    {
        try
        {
            //Verifica o OP Code passado no pacote
            switch (TipoPacote)
            {
                case CONSTANTES.TiposPacotesDadosEnum.RequisicaoLogin:
                    retorno = new
                    RequisicaoLoginController().RequisicaoLogin(JSON_Logic.Deserializa<RequisicaoLogin>(dados_pacote[1]));
                    break;
                case CONSTANTES.TiposPacotesDadosEnum.ReceberNovasMensagens:
                    retorno = JSON_Logic.Serializa<List<Mensagem>>(new
                    MensagemController().ReceberNovasMensagens(Convert.ToInt32(dados_pacote[1])));
                    break;
                case CONSTANTES.TiposPacotesDadosEnum.EnviaNovaMensagem:
                    retorno = new
                    MensagemController().EnviaNovaMensagem(JSON_Logic.Deserializa<Mensagem>(dados_pacote[1]));
            }
        }
        catch { }
    }
}

```

```

        break;
        case CONSTANTES.TiposPacotesDadosEnum.StatusContato:
            retorno = new
ContatoController().StatusContato(Convert.ToInt32(dados_pacote[1]));
            break;
        case CONSTANTES.TiposPacotesDadosEnum.StatusMensagem:
            retorno = new
MensagemController().StatusMensagem(JSON_Logic.Deserializa<Mensagem>(dados_pacote[1])).ToString();
            break;
        case CONSTANTES.TiposPacotesDadosEnum.NovoCadastro:
            retorno = new
UsuarioController().NovoCadastro(JSON_Logic.Deserializa<Usuario>(dados_pacote[1]));
            break;
        case CONSTANTES.TiposPacotesDadosEnum.EnviaNovoUsuario:
            retorno = new
UsuarioController().EnviaNovoUsuario(JSON_Logic.Deserializa<UsuarioAdicionado>(dados_pacote
[1]));
            break;
        case CONSTANTES.TiposPacotesDadosEnum.EnviaNovoGrupo:
            retorno = new
UsuarioController().EnviaNovoGrupo(JSON_Logic.Deserializa<Grupo>(dados_pacote[1]));
            break;
        case CONSTANTES.TiposPacotesDadosEnum.BuscaUsuarioPeloEmail:
            retorno = JSON_Logic.Serializa<Usuario>(new
UsuarioController().Busca(dados_pacote[1]));
            break;
        case CONSTANTES.TiposPacotesDadosEnum.BuscaUsuarioPeloID:
            retorno = JSON_Logic.Serializa<Usuario>(new
UsuarioController().Busca(Convert.ToInt32(dados_pacote[1])));
            break;
        case CONSTANTES.TiposPacotesDadosEnum.BuscaContato:
            retorno = JSON_Logic.Serializa<Contato>(new
ContatoController().Busca(Convert.ToInt32(dados_pacote[1])));
            break;
        default://Se o OP Code não estiver listado acima, está errado
            throw new NotImplementedException();
    }
    return retorno;
}
catch(Exception ex)
{
    return ex.Message;
}
}
else//Se não conseguiu encontrar um OP Code da primeira parte do pacote, lança exceção
{
    throw new InvalidCastException();
}
}

```

Quadro 7 – Método TrataPacote

Fonte: Própria

#### 4.5.4 Persistência dos dados

Para o protótipo, foi criado, no SQL Server, o banco de dados QTCC, e nele foram criadas as seguintes tabelas, também listadas na Figura 55:

- tbContato: Perfil-base de grupos e usuários (Código, Nome, Foto e se trata de um contato desativado)
- tbUsuario: Dados detalhados sobre um usuário (E-Mail, Senha e Texto de Perfil)

- **tbGrupo**: Usuário administrador de um grupo
- **tbListaContatos**: Lista de contatos de um usuário ou membros de um grupo
- **tmpMensagensPendentes**: Dados sobre mensagens que ainda não foram enviadas aos destinatários
- **tmpUsuariosLogados**: Registro de última vez em que um usuário foi visto online

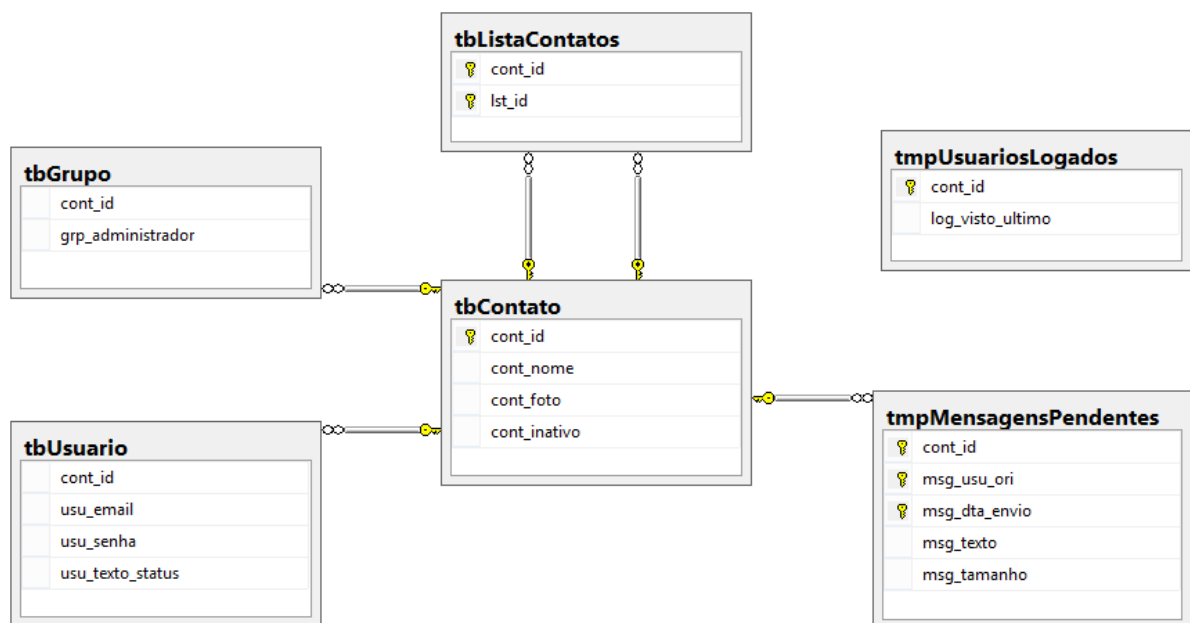


Figura 55 – Diagrama Entidade-Relacionamento (DER) do banco QTCC  
Fonte: Própria

Na tela inicial da aplicação servidor, existe o botão “Configurar Conexão”. Ao clicar nesse botão, é perguntado ao usuário se ele deseja reconfigurar o acesso ao banco de dados. Se confirmado, é aberta a tela de reconfigurar acesso à base de dados, como pode ser vista na Figura 56. Caso a aplicação servidor não tenha o acesso ao banco de dados configurado, a tela será aberta ao executar o QTCC\_Server.

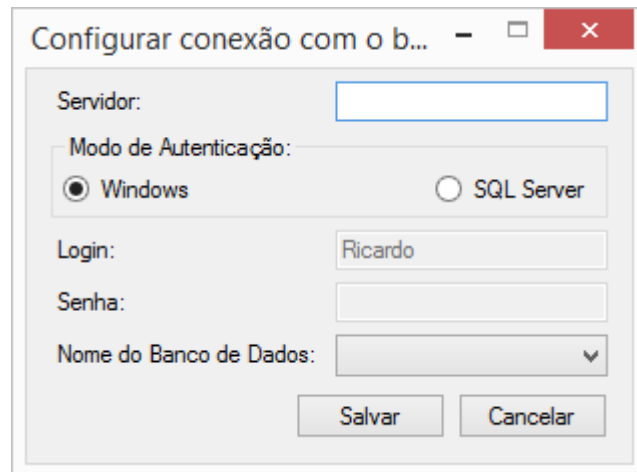


Figura 56 – Tela de reconfigurar acesso à base de dados  
Fonte: Própria

Ao preencher os campos “Servidor”, “Modo de Autenticação”, “Login” e “Senha” (caso não esteja sendo usada a autenticação pelo usuário atual no Windows), a aplicação gera uma conexão temporária e tenta listar os bancos de dados disponíveis na instância de SQL Server configurada. Caso não consiga, é exibida a mensagem de erro vista na Figura 57:

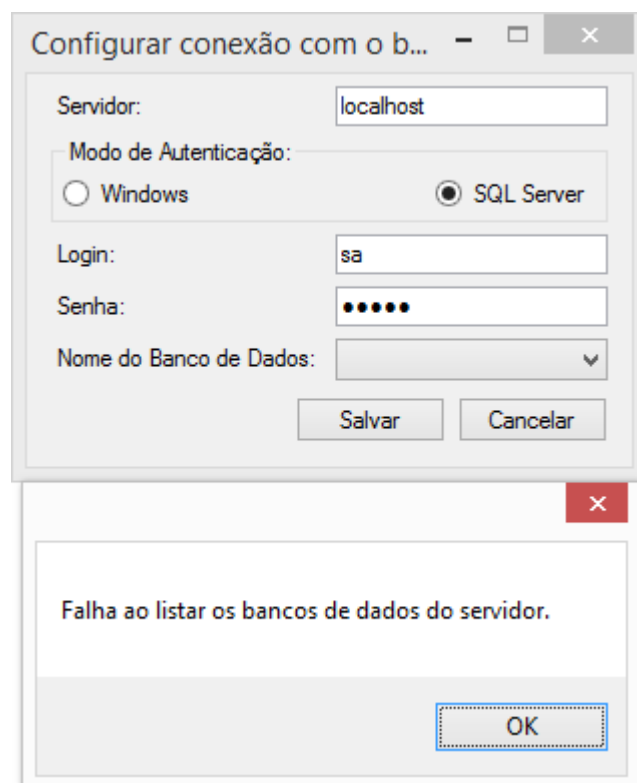


Figura 57 – Mensagem de erro ao realizar configuração incorreta de conexão  
Fonte: Própria



A tela de reconfigurar o acesso à base de dados pertence à outro projeto dentro da solução da aplicação servidor, chamado CFG\_BD\_XML. Essa aplicação secundária tem como propósito realizar o intermédio entre a aplicação principal e o banco de dados, provendo classes e métodos de acesso à base de dados configurada. Seu diagrama de classes está representado na Figura 58:

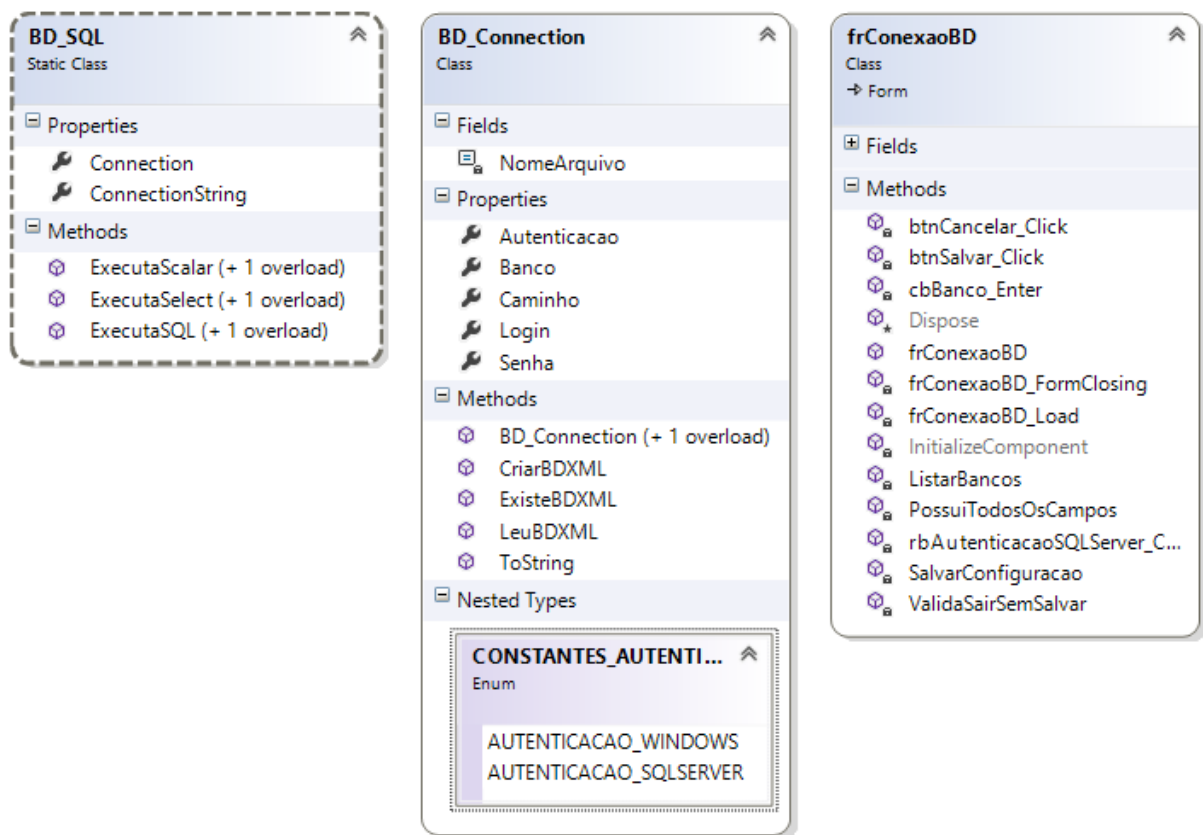


Figura 58 – Diagrama de classes da aplicação CFG\_BD\_XML  
Fonte: Própria

Essa aplicação salva os dados da conexão configurada em um arquivo XML chamado BD.XML. Para isso, ela serializa o objeto, do tipo BD\_Connection, para XML através do método CriarBDXML, ilustrado no Quadro 8:

```

/// <summary>
/// Método responsável por gerar o arquivo BD.XML na pasta do programa
/// </summary>
/// <param name="connection">A Connection String na forma de objeto do tipo
BD_Connection</param>
public static void CriarBDXML(BD_Connection connection)
{
    //Inicializa o serializador XML com base no tipo do objeto "connection"
    XmlSerializer writer = new XmlSerializer(connection.GetType());
    //Inicializa o objeto responsável por gravar o arquivo "BD.XML"
    StreamWriter arquivo = new StreamWriter(NomeArquivo);
    try
    {
        //Serializa a Connection String em XML
        writer.Serialize(arquivo, connection);
    }
    finally
    {
        //Fecha o buffer de "arquivo"
        arquivo.Close();
    }
}

```

Quadro 8 – Método CriarBDXML  
Fonte: Própria

Para realizar o acesso ao banco de dados na aplicação servidor, foi necessário criar classes DAO (*Data Access Object*) que incluíssem o *namespace* CFG\_BD\_XML. Essas classes são estáticas e possuem apenas os métodos necessários na aplicação.

Um exemplo de classe de acesso à dado é a classe ContatoDAO, responsável por realizar o acesso à tabela tbContato. Esta classe possui o método estático AdicionaContatoListaContatos, que, a partir do código do usuário e do código do contato, o adiciona à lista de contatos do usuário e retorna o sucesso ou não da ação, como pode ser visto no Quadro 9:

```

/// <summary>
/// Método responsável por adicionar um contato à lista de contatos de outro contato
/// </summary>
/// <param name="cont_id">Contato que irá adicionar um contato à lista (Usuário ou
grupo)</param>
/// <param name="lst_id">Contato a ser adicionado à lista (Usuário ou Grupo)</param>
/// <returns>Se a inserção do contato à lista de contatos foi concluída com
sucesso</returns>
public static bool AdicionaContatoListaContatos(int cont_id, int lst_id)
{
    bool retorno = false;
    //Inicializa a conexão com base na Connection String utilizada na aplicação inteira
    SqlConnection c = BD_SQL.Connection;
    try
    {
        c.Open();
        //Construção de um comando SQL parametrizado. Previne contra SQL Injection
        SqlCommand cmd = new SqlCommand("insert into
tbListaContatos(cont_id,lst_id)values(@Cont_Id,@Lst_Id)",c);
        cmd.Parameters.AddWithValue("@Cont Id", cont_id);
        cmd.Parameters.AddWithValue("@Lst Id", lst_id);
        //Retorna se a quantidade de registros afetados (inseridos, no caso) foi maior que
zero
        retorno = BD_SQL.ExecutaSQL(cmd) > 0;
    }
}

```

```
finally
{
    //Fecha a conexão
    c.Close();
}
return retorno;
```

Quadro 9 – Método estático AdicionaContatoListContatos  
Fonte: Própria

## 5 CONCLUSÃO

Este trabalho apresentou um pouco da história e evolução dos sistemas operacionais e plataformas computacionais. Apresentou os conceitos de desenvolvimento multiplataforma, assim como algumas ferramentas que auxiliam nesse tipo de desenvolvimento, sendo uma delas o objeto de estudo deste trabalho, o Qt. Foi apresentado também um estudo de caso, tratando do desenvolvimento de um protótipo de trocas de mensagens com a ferramenta.

Embora tenham ocorrido alguns problemas durante o desenvolvimento do protótipo, o objetivo geral, que visava avaliar a viabilidade do desenvolvimento multiplataforma utilizando-se o Qt, assim como verificar a necessidade ou não de refatoração de trechos do código, foi alcançado. O protótipo desenvolvido, sem necessidade de refatoração alguma no código, funcionou de maneira praticamente idêntica em ambas as plataformas. Mesmo apresentando alguns problemas em componentes para a plataforma Android, o desenvolvimento multiplataforma se mostrou viável no caso de uma aplicação de comunicação via TCP, com a utilização do *framework* Qt.

A documentação disponibilizada pelo próprio Qt Project é bastante vasta, incluindo descrições detalhadas de grande parte dos métodos e classes existentes, incluindo vários exemplos de utilização. Existem também livros com grande riqueza de conteúdo, tratando de muitos aspectos do *framework*. Isso possibilita que desenvolvedores iniciantes possam dominar com bastante facilidade o desenvolvimento utilizando o Qt, mesmo não conhecendo todos os conceitos por trás do desenvolvimento multiplataforma. Embora não seja obrigatório, é aconselhável possuir conhecimentos prévios sobre a linguagem C++, utilizada pelo *framework*.

De maneira geral, o *framework* Qt cumpre o que promete, e no estudo apresentado facilitou o desenvolvimento multiplataforma, para as plataformas Windows, Linux e Android, e em sua versão gratuita apresenta um *kit* bastante completo, suficiente para desenvolver aplicativos para muitos sistemas operacionais.

## 6 TRABALHOS FUTUROS

Durante esse trabalho de conclusão de curso foi abordado o estudo de ferramentas para desenvolvimento multiplataforma, e foi desenvolvido um protótipo utilizando-se uma delas, o *framework* Qt. Verificou-se que esse *framework* apresenta diversas características que tornam possível o desenvolvimento de aplicações bastante complexas que possam rodar em variados sistemas operacionais.

Dessa forma o assunto abordado neste trabalho apresenta possibilidades para complementos ou até mesmo servindo como base para novas linhas de pesquisa, sendo algumas ideias:

- Implementação de novas funcionalidades, como envio de imagens, vídeos e áudio;
- Implementar as funcionalidades de grupo, como criação, administração e troca de mensagens;
- Recriar as interfaces do sistema utilizando-se para isso o QML, para estudo de sua responsividade em plataformas *mobile*, principalmente;
- Implementar a parte servidor do sistema também na plataforma Qt, ao invés de se usar o C#, como feito neste trabalho;
- Testar aplicações em demais plataformas, como Windows Phone e plataformas Apple, para demonstrar a necessidade ou não da refatoração do código, visto que entre Windows, Linux e Android os resultados mostraram necessidade de refatoração próxima a zero.

## 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 Como Programar, Oitava Edição*. 2009. Disponível em: <[http://cefsa-fft.bv3.digitalpages.com.br/users/publications/9788576055631/pages/\\_1](http://cefsa-fft.bv3.digitalpages.com.br/users/publications/9788576055631/pages/_1)>. Acesso em 12 de novembro 2014.

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

HORSTMANN, C; CORNELL, G; *Core Java Volume 1 – Fundamentos, Oitava Edição*. 2009. Disponível em: <[http://cefsa-fft.bv3.digitalpages.com.br/users/publications/9788576053576/pages/\\_1](http://cefsa-fft.bv3.digitalpages.com.br/users/publications/9788576053576/pages/_1)>. Acesso em 22 novembro 2014.

JAVA. *Obtenha informações sobre a Tecnologia Java*. 2014. Disponível em: <[https://www.java.com/pt\\_BR/about/](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](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](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. *Nokia to acquire Trolltech*. 2014a. Disponível em: <<http://blog.qt.digia.com/blog/2008/01/28/nokia-to-acquire-trolltech/>>. Acesso em 14 dezembro 2014.

\_\_\_\_\_. *Digia extends its commitment to Qt with plans to acquire full Qt software technology and business From Nokia*. 2014b. Disponível em: <<http://blog.qt.digia.com/blog/2012/08/09/digia-extends-its-commitment-to-qt-with-plans-to-acquire-full-qt-software-technology-and-business-from-nokia-4/>>. Acesso em 14 dezembro 2014.

\_\_\_\_\_. *Qt 4.8.x Support to be Extended for Another Year*. 2014c. Disponível em: <<http://blog.qt.digia.com/blog/2014/11/27/qt-4-8-x-support-to-be-extended-for-another-year/>>. Acesso em 14 dezembro 2014.

QT IO. *Download Qt*. 2014a. Disponível em: <<http://www.qt.io/download>>. Acesso em 05 setembro 2014.

\_\_\_\_\_. *Qt in Use*. 2014b. Disponível em: <<http://www.qt.io/qt-in-use>>. Acesso em 22 novembro 2014.

\_\_\_\_\_. *Qt 5.4*. 2014c. Disponível em: <<http://www.qt.io/qt5-4/>>. Acesso em 13 dezembro 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.

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

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

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

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

\_\_\_\_\_*New\_Signal\_Slot\_Syntax*. 2014o. Disponível em: <[http://qt-project.org/wiki/New\\_Signal\\_Slot\\_Syntax](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/](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.

\_\_\_\_\_*Xamarin Store*. 2014i. Disponível em: <<https://store.xamarin.com/>>. Acesso em 26 novembro 2014.

## **APÊNDICE A: PROTÓTIPO DA APLICAÇÃO**

O código-fonte completo do protótipo está disponível em mídia digital que acompanha este trabalho.