UNIVERSIDADE FEDERAL DE SANTA CATARINA

CENTRO TECNOLÓGICO

PROGRAMA DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

MARGE: FRAMEWORK PARA DESENVOLVIMENTO DE DE APLICAÇÕES EM JAVA QUE FAÇAM USO DA TECNOLOGIA BLUETOOTH

BRUNO CAVALER GHISI

FLORIANÓPOLIS

JULHO DE 2007

BRUNO CAVALER GHISI

MARGE: FRAMEWORK PARA DESENVOLVIMENTO DE DE APLICAÇÕES EM JAVA QUE FAÇAM USO DA TECNOLOGIA BLUETOOTH

Trabalho de Conclusão de Curso apresentada no programa de Graduação no Centro Tecnológico da Universidade Federal de Santa Catarina, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Prof. Frank Siqueira, Dr. - Orientador

MARGE: FRAMEWORK PARA DESENVOLVIMENTO DE
DE APLICAÇÕES EM JAVA QUE FAÇAM USO DA
TECNOLOGIA BLUETOOTH

Por

BRUNO CAVALER GHISI

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina, Programa de Graduação em Sistemas de Informação, para obtenção do grau de Bacharel em Sistemas de Informação, pela Banca Examinadora, formada por:

____

Presidente: Prof. Frank Siqueira, Dr. – Orientador, UFSC

____

Membro: Prof. Leandro José Komosinski, Dr., UFSC

____

Membro: Prof. José Eduardo De Lucca, Dr., UFSC

____

Membro: Prof. Olinto José Varela Furtado, Dr., UFSC

Dedico esse trabalho aos meus pais.

AGRADECIMENTOS

Meus sinceros agradecimentos aos verdadeiros mestres, meus pais, Cícero e Eliane, por todo incentivo e ensinamentos durante essa longa jornada que é a vida.

A minha irmã, Júlia, e aos meus familiares, por serem parte constante da minha felicidade.

A minha namorada, Amanda, por apoiar o meu viver nesse mundo louco, chamado por ela de “Mundo Java”.

Ao meu orientador, Prof. Frank Siqueira, por toda a ajuda durante o desenvolvimento desse trabalho e por servir de referência nessa minha caminhada.

Aos grandes mestres da UFSC e a toda banca avaliadora desse trabalho, por representarem aquilo que um dia quero ser profissionalmente.

A comunidade Mobile & Embedded do java.net e aos grandes amigos que conheci em todos esses eventos da computação nos últimos tempos, por todo o suporte e incentivo acerca do projeto.

E, finalmente, aos meus amigos da universidade, as minhas duas turmas queridas, 031 e 032, aos meus amigos que não são da área tecnológica e a todas essas pessoas com quem compartilho meus momentos de lazer e estudo, vocês são incríveis.

Muito obrigado!

“You have to generate many ideas and then you have to work very hard only to discover that they don’t work. And you keep doing that over and over until you find one that does work”.

John Warner Backus, FORTRAN inventor

RESUMO

Bluetooth é uma tecnologia sem fio muito usada para conectar dispositivos a curta distância, que ganhou bastante espaço nos últimos anos por estar disponível na maioria dos celulares e *notebooks* mais recentes. A linguagem de programação Java, atualmente, também aparece como uma das principais para o desenvolvimento de aplicações em vários dispositivos, devido as diversas vantagens que a mesma possui. Java possui uma especificação para uso de Bluetooth, definida pela JSR 82, mas esta, por conseguinte, possui um difícil aprendizado. Sendo assim, neste trabalho foi proposto o *framework* Marge com o intuito de facilitar o desenvolvimento de aplicações Java que façam uso de Bluetooth. O Marge é um projeto de código livre disponível no portal java.net e pode ser usado tanto na plataforma Java ME, quanto na Java SE. Para validar o seu uso nessas duas tecnologias, foram implementadas algumas aplicações exemplos também.

Palavras-chave: Bluetooth. Java. JSR 82. Marge.

ABSTRACT

Bluetooth is a wireless technology mostly used to connect devices in a short range, which became so famous in the last years due to it is being avaiable in the majority of recent notebooks and cellphones. The programming language Java, nowdays, appears, as well, as one of the main to develop applications in different devices, because of the inumerous advantages that it has. Java has a specification to the use of Bluetooth, defined by JSR 82, but this one has a difficult learning curve. So, in this project was proposed the Marge framework that is indeed to facilitate the development of Java applications using Bluetooth technology. Marge is an open source project hosted on java.net and it can be used in Java ME plataform, as well as in Java SE. To validate its use in both of these two technologies, some demo applications were implemented too.

Key-words: Bluetooth. Java. JSR 82. Marge.

LISTA DE ILUSTRAÇÕES

Figura 2.1 – Logo Bluetooth (BLUETOOTH SIG, 2006).................................28
Figura 2.2 – *Piconet* (TALLARICO, 2002)...30
Figura 2.3 – *Scatternet* (TALLARICO, 2002)...31
Figura 2.4 – *Service Discovery DataBase* (ORTIZ, 2006)....................................32
Figura 2.5 – Pilha de Protocolos Bluetooth (APPLE, 2006b)............................35
Figura 2.6 – Perfis Bluetooth (APPLE, 2006b)...39
Figura 3.1 - Plataforma Java (SUN, 2006b)..48
Figura 3.2 - Arquitetura do Java SE e suas tecnologias (SUN, 2006b)..............49
Figura 3.3 - Estrutura da tecnologia Jave ME com suas configurações, perfis e pacotes adicionais (SUN, 2006a)..50
Figura 3.4 – Configurações do Java ME (SUN, 2006e).....................................51
Figura 3.5 – Relacionamento do CDLC perante o CDC (SUN, 2006e).............51
Figura 3.6 - Estrutura dos perfis suportados pela CDC e CLDC (SUN, 2006e).53
Figura 3.7 - Estrutura mostrando as configurações, perfis, pacotes opcionais e APIs dos fabricantes (SUN, 2006e)...54
Figura 5.1 – Arquitetura onde o Marge está inserido...63
Figura 5.2 – Logo atual do projeto Marge...63
Figura 5.3 – Diagrama de classes do pacote *net.java.dev.marge.communication*..66
Figura 5.4 – Diagrama de classes do pacote *net.java.dev.marge.entity*.............69
Figura 5.5 – Diagrama de classes do pacote *net.java.dev.marge.entity.config*...67
Figura 5.6 – Diagrama de classes do pacote *net.java.dev.marge.factory*............68
Figura 5.7 – Diagrama de classes do pacote *net.java.dev.marge.inquiry*..........69
Figura 5.8 – Diagrama de classes da versão 0.4.0 do Marge.............................70
Figura 5.9 – Diagrama de sequência simplificado do caso mais comum do

processo de uso de um servidor RFCOMM..73
Figura 5.10 – Exemplo de código do Cliente..74
Figura 5.11 – Diagrama de sequência simplificado do caso mais comum do
processo de uso de um cliente RFCOMM..75
Figura 5.12 – Exemplo de código do Servidor..76
Figura 5.13 – Interface gráfica do chat..77
Figura 5.14 – Interface gráfica do Jogo da Velha..78
Figura 5.15 – Interface gráfica do Pong...78
Figura 5.16 – J2ME Wireless Toolkit 2.2...80
Figura 5.17 – Nokia 6230..80
Figura 5.18 – Nokia 6111..81
Figura 5.19 – Nokia 6681..81
Figura 5.20 – Sony Ericsson W300..81
Figura 5.21 – Sony Ericsson W800..82

LISTA DE TABELAS

Tabela 2.1 – Identificadores dos serviços mais comuns e seus tipos (adaptado de BLUETOOTH MEMBERSHIP, 2006)..33

Tabela 3.1 - Pacotes da JSR 82 (adaptado de SUN, 2006e)................................55

Tabela 3.2 – Implementações da JSR 82 para uso em computadores (adaptado de JAVA BLUETOOTH, 2006).. 56

LISTA DE ABREVIATURAS E SIGLAS

API - Application Programming Interface

CDC - Connected Device Configuration

CLDC - Connected, Limited Device Configuration

DUN - Dial-Up Networking

FH-CDMA - Frequency Hopping - Code-Division Multiple Access

FTP - File Tranfer Profile

GC – Garbage Collector

GAP - Generic Access Profile

GFC - Generic Connection Framework

GHz - GigaHertz

GIAC - General Inquiry Access Code

GUI - Graphical User Interface

HCI - Host Controller Interface

HID - Human Interface Device

HRCP - Hardcopy Cable Replacement Profile

HSP - Headset Profile

IAC - Inquiry Access Code

IDE – Integrated Development Environment

IoC – Inversion of Control

IrOBEX - Infrared OBEX

ISM - Industrial, Scientific and Medicine

LGPL – GNU Lesser General Public License

PDA - Personal Digital Assistants

Java EE - Java Enterprise Edition

Java ME - Java Micro Edition

Java SE - Java Standard Edition

JAD - Java Application Descriptor

JAR - Java Archive

JCP - Java Community Process

JSR - Java Specification Request

JVM - Java Virtual Machine

KB - Kilobyte(s)

KVM - Kilobyte Virtual Machine

L2CAP - Logical Link Control and Adaptation Protocol

LBS - Location Based Services

LIAC - Limited Inquiry Access Code

MB - Megabyte(s)

MIDP - Mobile Information Device Profile

OBEX - Object Exchange

ORB – Object Request Broker

OS - Operational System

PIN – Personal Identification Number

RAM - Random Access Memory

RFCOMM - RS-232 Serial Cable Emulation Profile

RMI – Remote Method Invocation

RMS – Record Management System

SDA - Service Discovery Application

SDDB - Service Discovery DataBase

SDP - Service Discovery Protocol

SIG - Special Interest Group

SPP - Serial Port Profile

SYNC - Synchronization Profile

UNO – Universal Network Objects

USB - Universal Serial Bus

UUID - Universally Unique IDentifier

VM - Virtual Machine

WTK – Wireless Toolkit

XML – Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO..21

1.1 OBJETIVOS..23

1.2 JUSTIFICATIVA..24

1.3 ESCOPO..25

1.3 ORGANIZAÇÃO DO TEXTO..25

2 BLUETOOTH..27

2.1 ORIGEM DO NOME E LOGO..28

2.2 INFRA-ESTRUTURA..28

2.3 PICONET E SCATTERNET..29

2.4 PESQUISA POR DISPOSITIVOS E SERVIÇOS..31

2.5 SERVIÇOS BLUETOOTH..32

2.6 ARQUITETURA..34

2.6.1 Camadas Inferiores..35

2.6.2 Camada de Comunicação..36

2.6.3 Camadas Superiores..37

2.7 PERFIS BLUETOOTH..38

2.8 SEGURANÇA..41

2.8.1 Modos de Segurança..42

2.8.2 Autenticação (Pareammento e Ligação)..43

2.8.3 Autorização..44

2.8.4 Encriptação..44

2.8.5 Gerenciador de Segurança..45

2.8.6 Ataques à Dispositivos..46

3 JAVA..48

3.1 JAVA PLATAFORM, STANDARD EDITION (Java SE)...........................48

3.2 JAVA PLATAFORM, MICRO EDITION (Java ME)..................................49

3.2.1 Configurações..51

3.2.2 Generic Connection Framework...52

3.2.3 Perfis...52

3.2.4 Pacotes Opcionais...54

3.3 JAVA APIS FOR BLUETOOTH (JSR 82)..55

3.4 IMPLEMENTAÇÕES DA JSR 82 55

3.8 JAVA VIRTUAL MACHINE...56

4 FRAMEWORKS..58

4.1 CARACTERÍSTICAS..58

4.2 CLASSIFICAÇÃO...59

4. DESAFIOS RELACIONADOS...60

5 PROJETO MARGE...62

5.1 METODOLOGIA DE DESENVOLVIMENTO 64

5.2 FERRAMENTAS DE DESENVOLVIMENTO...64

5.3 CLASSIFICAÇÃO DO FRAMEWORK 65

5.4 ESTRUTURA DO FRAMEWORK 65

5.4.1 Pacotes e classes...65

5.4.2 Diagrama de classes..70

5.5 DOCUMENTAÇÃO..70

5.5.1 Javadoc...71

5.5.2 Formas de utilização do Framework...71

5.6 APLICAÇÕES DESENVOLVIDAS...76

5.6.1 Chat (*Blue Chat*) 76

5.6.2 Jogo da Velha (*Tik Tak Tooth*)..77

5.6.3 Pong (*Blue Pong*)...78

5.6.4 Passador de slides para o Impress da suite Open Office 78

5.7 TESTES.................. 79

5.7.1 Emuladores...79

5.7.2 Celulares..80

5.7.1 Computadores 82

6 CONCLUSÃO.... 83

6.1 CONSIDERAÇÕES FINAIS...83

6.2 TRABALHOS FUTUROS 84

7 REFERÊNCIAS BIBLIOGRÁFICAS 85

ANEXOS 93

ANEXO A - DOCUMENTOS 93

A.1 Artigo do Projeto Marge 93

ANEXO B - REFERÊNCIAS AO PROJETO 106

B.1 Destaque em vermelho sobre a entrevista do projeto Marge, realizada durante o FISL 8.0, na página principal do java.net (http://mobileandembedded.org), que foi postada no dia 2 de Maio de 2007. 106

B.2 Destaque em vermelho sobre a entrevista do projeto Marge, realizada durante o FISL 8.0, na página da comunidade Mobile & Embedded do java.net (http://mobileandembedded.org), que foi postada no dia 1 de Maio de 2007. 106

B.3 Destaque em vermelho na página da entrevista do Marge, realizada durante o FISL 8.0 (http://today.java.net/pub/a/today/2007/05/01/mobileandembedded-podcast2.html), que foi postada no java.net dia 1 de Maio de 2007.................107

B.4 Destaque em vermelho do *post* sobre o Marge da Sue Abellera (http://weblogs.java.net/blog/sue_abellera/archive/2007/04/my_highlights_f_1.html), ex-gerente do desenvolvimento da máquina virtual Java ME, no dia 29 de Abril, em seu blog no java.net...108

B.5 Destaque em vermelho sobre o *post* do Marge, no dia 23 de Abril, no blog do Neto Marin (http://netomarin.blogspot.com/), um dos criadores do JME Brasil (http://www.jmebrasil.org)..108

B.6 Destaque em vermelho do projeto Marge que venceu a categoria "Best Video – Open Source Techonology" do concurso Java Mobile Application Video Contest (http://java.sun.com/javame/contest/index.jsp) promovido pela comunidade Mobile & Embedded do java.net, no dia 10 e Maio. 109

B.7 Destaque em vermelho para a entrevista do projeto Marge no blog da empresa irlandesa, que atua nessa área de Bluetooth e Java, Rococo (http://www.rococosoft.com/weblog/archives/2007/05/cool_projects_using_jsr82_1_ma.html), no dia 14 de Maio...110

ANEXO C - CÓDIGO FONTE DO MARGE 0.4.0.......................................112

C.1 net.java.dev.marge.communication.CommunicationChannel...................112

C.2 net.java.dev.marge.communication.CommunicationListener...................113

C.3 net.java.dev.marge.communication.ConnectionListener...........................115

C.4 net.java.dev.marge.communication.L2CAPCommunicationChannel........116

C.5 et.java.dev.marge.communication.RFCOMMCommunicationChannel.....119

C.6 net.java.dev.marge.entity.ClientDevice..122

C.7 net.java.dev.marge.entity.Device..123

C.8 net.java.dev.marge.entity.ServerDevice..127

C.9 net.java.dev.marge.entity.config.ClientConfiguration...............................128

C.10 net.java.dev.marge.entity.config.Configuration.......................................129

C.11 net.java.dev.marge.entity.config.MargeDefaults.....................................131

C.12 net.java.dev.marge.entity.config.ServerConfiguration............................132

C.13 net.java.dev.marge.factory.CommunicationFactory................................137

C.14 net.java.dev.marge.factory.LCAPCommunicationFactory.......................138

C.15 net.java.dev.marge.factory.RFCOMMCommunicationFactory..............142

C.16 net.java.dev.marge.inquiry.DefaultDiscoveryListener............................145

C.17 net.java.dev.marge.inquiry.DeviceDiscoverer..149

C.18 net.java.dev.marge.inquiry.InquiryListener...151

C.19 net.java.dev.marge.inquiry.ServiceDiscoverer..152

C.20 net.java.dev.marge.inquiry.ServiceSearchListener..................................154

ANEXO D - CÓDIGO FONTE DO BLUE CHAT USANDO MARGE 0.4.0...156

D.1 net.java.dev.marge.chat.ChatMIDlet..156

D.2 net.java.dev.marge.chat.ui.AboutScreen...157

D.3 net.java.dev.marge.chat.ui.ChatRoom..158

D.4 net.java.dev.marge.chat.ui.InquiryScreen...160

D.5 net.java.dev.marge.chat.ui.LoadingScreen..163

D.6 net.java.dev.marge.chat.ui.MainMenu..164

D.7 net.java.dev.marge.util.ImageUtil..167

1 INTRODUÇÃO

Tecnologias sem fio se tornaram muito populares no mundo devido a sua praticidade e comodidade. Uma grande quantidade delas surge de tempos em tempos para suprir diferentes necessidades. Dessa forma, também surgiu a tecnologia Bluetooth, que veio inicialmente com o intuito de eliminar a necessidade de utilizar cabos elétricos para conexão de equipamentos. A primeira concepção dessa tecnologia foi criada em 1994 com o objetivo de fazer a conexão e troca de mensagens entre dispositivos a curta distância. Hoje em dia, essa distância já se limita a até 100 metros e os *chips* Bluetooth - devido aos preços estarem se tornando cada vez menores – já são integrados em diversos dispositivos, tais como: impressoras, *notebooks*, câmeras, fones de ouvidos, televisões e principalmente em telefones celulares. O Bluetooth SIG anunciou, no final de 2006, que as vendas de dispositivos Bluetooth já ultrapassavam a marca de 1 bilhão, o que é um número muito maior que a quantidade de computadores no mundo. Além disso, ainda foi confirmado que a maioria desses *chips* está sendo utilizada em telefones celulares, o que faz com que toda essa projeção de crescimento seja muito maior, em virtude da grande adoção do uso de celulares, que até o fim de 2007, segundo estimativas, somarão mais de 3 bilhões no mundo (CNET NEWS, 2007).

Junto a esse crescimento está também o da tecnologia Java, que atualmente é amplamente usada pela comunidade computacional e pelas empresas, grande parte devido ao poder no que diz respeito a portabilidade. Java possui uma linguagem de programação poderosa e seus programas rodam em uma variedade de dispositivos, fator que também contribuiu para que ocorresse uma crescimento significativo no desenvolvimento de celulares com suporte à ela.

Para permitir a integração entre as tecnologias Bluetooth e Java foi criada a *Java APIs for Bluetooth,* conhecida por JSR 82. A JSR 82 atualmente está disponível em diversos dispositivos e define a especificação para comunicação Bluetooth em diversos níveis. Com toda essa facilidade de suporte por parte da tecnologia Java, e com a grande adoção do Bluetooth pelos fabricantes, se faz possível, assim, a criação de diversas aplicações interessantes e/ou de dispositivos que utilizam essa tecnologia para cenários distintos, tais como em casa, no trabalho, no lazer ou qualquer outro ambiente que requeira mobilidade.

Alguns exemplos desse cenários, segundo o SIG do Bluetooth, seriam:

- a sincronização de dispositivos variados para fins diversos (desde compartilhamento da agenda até download de notícias);
- exibir imagens contidas em um celular em uma televisão/monitor com a finalidade de fazer uma apresentação;
- imprimir arquivos em uma impressora sem a necessidade de conectar um cabo;
- utilização de acessórios sem fio com o objetivo de deixar o ambiente de trabalho mais limpo e assim poder aumentar a produtividade, como fez o Fedex (EWALT, 2006) e a UPS (MOBILE INFO, 2006);
- praticidade com o advento dos fones de ouvido sem fio via Bluetooth, popularmente chamados de Hands Free;
- utilização do celular para fazer ligações pelo telefone fixo via Bluetooth, como o produto oferecido pela BrasilTelecom (ESTADAO, 2006);
- interação com um viva voz no carro para possibilitar conversas enquanto se dirige, como o Fiat Stilo que já suporta isso (MUNDO SEM FIO, 2006);
- alguma jaqueta que permita ouvir música e trocar faixas enquanto se pratica alguma atividade que utiliza as duas mãos, tal como escalar uma montanha ou correr;
- óculos adaptados com Hands Free, última criação da empresa Oakley em parceria com a Motorola (PHONEY WORLD, 2006);
- utilizar o computador para permitir navegação de celulares na Internet via Bluetooth em, por exemplo, algum *cyber* café;
- *broadcast* de mensagens, imagens ou vídeos para fins de marketing - um tipo especial de mobile marketing - como faz a empresa BlueCasting, que já ofereceu serviços a grandes empresas como a Bristish Airways, Carling, Volvo, Nokia, Land Rover e que também fez a divulgação do novo álbum da banda Coldplay no mêtro de Londres, onde foram encontrados cerca de 87 mil dispositivos com Bluetooth ligado e 13 mil receberam conteúdo durante uma campanha de duas semanas (BLUECASTING, 2006);
- outros casos de sucesso criados pela empresa BlipSystem que utilizou de posters de propagandas que interagiam com o consumidor no lançamento do filme ScaryMovie4 (lançado no Brasil com o título 'Todo Mundo em Pânico 4'), além da implantação de

um sistema em um shopping center na África do Sul na qual as lojas ofereciam promoções às pessoas via Bluetooth no celular (BLIPSYSTEMS, 2006),;

- um ônibus ou trem utilize o Bluetooth para avisá-lo que seu ponto está próximo, implementado pelo governo Finlandês com o projeto Noppa (NOPPA, 2006);
- encontro de pessoas com mesmo interesses que você através do Bluetooth com intuito de propor novos paradigmas de redes sociais baseados em localização (chamado também de LBS);
- uma caneta que facilita a compilação e digitalização das idéias durante um processo de *brainstorming* em uma reunião de executivos (EBEAM PROJECTION, 2006);
- televisão com Bluetooth (SAMSUNG, 2006);
- jogos multiplayers via Bluetooth;
- utilização do celular para poder fazer pedido em uma máquina de refrigerantes ou em um restaurante;

e tantas outras idéias de serviços e produtos que estão surgindo e que surgirão - à medida que ocorre o avanço da tecnologia - para tornar a vida humana mais fácil, interessante e proporcionar às empresas novas formas de efetuar negócios.

A partir desse cenário que estamos vivenciando, o presente trabalho estará focado na criação de um *framework*, intitulado Marge, que irá prover uma camada para facilitar o desenvolvimento de aplicações Java que façam uso da tecnologia Bluetooth através da JSR 82. Com isso, poderá ser criado, de forma mais simples, aplicações para dispositivos que implementem essa especificação, como celulares e computadores pessoais.

1.1 Objetivos

Este trabalho tem como objetivo principal desenvolver um *framework* capaz de abstrair do programador a parte responsável pela comunicação Bluetooth em Java (definida pela JSR 82), de modo que o mesmo possua suporte aos diversos protocolos de comunicação que envolvem

a tecnologia, configurações, facilitando, assim, a criação de novos serviços e aplicativos.

Para atingir este objetivo, será necessário efetuar um estudo sobre a tecnologia Bluetooth focado no desenvolvimento de aplicativos nas plataformas Java ME (*Micro Edition*) e Java SE (*Standard Edition*). Em seguida, será realizado o projeto e a implementação do *framework*. Ao final disso, serão criados alguns aplicativos teste que validem o uso desse *framework*. Para desenvolver os aplicativos com a plataforma Java ME, será necessário utilizar, por exemplo, dispositivos celulares com Java que possuam Bluetooth e a JSR 82 disponível. Já com Java SE, em algum computador pessoal, será necessário um suporte Bluetooth em hardware nativo ou algum adaptador, por exemplo um adaptador USB (conhecido como *dongle*), além do mesmo possuir no sistema operacional em questão alguma implementação da pilha de protocolos Bluetooth e alguma biblioteca que implemente a JSR 82.

1.2 Justificativa

O Bluetooth vem caminhando para se tornar definitivamente a tecnologia padrão para conexão em curtas distâncias Em 1998, um conjunto de grandes empresas como a Ericsson, IBM, Intel, Toshiba, Nokia, entre outras, montaram o SIG do Bluetooth para definir uma especificação aberta para a tecnologia. Hoje em dia já se encontra disponível a versão 2.0 dessa especificação e atualmente existem mais de 8000 empresas que fazem parte desse grupo (BLUETOOTH SIG, 2006), sendo que ao longo desse tempo foram criados, por diversos fabricantes distintos de diversos equipamentos, mais de 3800 produtos qualificados (BLUETOOTH QUALIFICATION, 2006) que possuem algum suporte a essa tecnologia. Além disso, diversos produtos estão sendo desenvolvidos - como comentados anteriormente - com uma boa repercussão por parte da população e de empresas, o que faz crer na criação de um grande nicho de mercado.

Sendo assim, o trabalho pretende mostrar algumas das utilidades que a tecnologia Bluetooth pode oferecer e com isso implementar algumas aplicações de teste, usando o *framework*, que funcionem entre celulares e computadores pessoais. A idéia principal para a criação do

projeto surgiu pelo fato de que, atualmente, para se desenvolver algo utilizando Bluetooth com Java, necessita-se de um conhecimento relativamente grande antes de se poder começar a implementar algo, pois a especificação JSR 82 é bastante complexa. A idéia do *framework* é justamente reduzir essa curva de aprendizado e propor uma arquitetura comum para o desenvolvimento de aplicações com esse contexto.

1.3 Escopo

Embora o campo de uso de aplicações Bluetooth seja grande, esse trabalho se limitará apenas ao desenvolvimento do *framework*. Dessa mesma forma, não será intenção implementar a JSR 82 para nenhum sistema operacional, apesar de que muitas vezes, durante o desenvolvimento desse projeto, far-se-á necessário, em virtude do pouco suporte de domínio público que essa área contém, uma análise aprofundada de alguma dessas implementações com o objetivo de corrigir possíveis problemas que possam estar impedindo o andamento do trabalho.

Também não faz parte do escopo desse trabalho o descobrimento e a resolução de possíveis erros (*bugs*) relativos a algum uso do Bluetooth que algum modelo de celular possa conter. O mesmo se limitará apenas em reportá-lo ao fabricante do dispositivo.

1.4 Organização do Texto

O capítulo 2 apresenta a tecnologia Bluetooth, mostrando um pouco de suas caraterísticas técnicas e sua estrutura de protocolos e perfis.

No capítulo 3 será abordado a tecnologia Java, atendo o foco as plataformas Java SE e Java ME. Além disso, será introduzido o pacote opcional caracterizado pela especificação JSR 82, que é referente ao uso de Bluetooth em Java.

O capítulo 4 é dedicado a *frameworks*. Nele será discutido o que é um *framework* e será

exposto algumas de suas classificações.

Por fim, no capítulo 5, será detalhado o que é projeto Marge - que surgiu como uma idéia prática desse trabalho - e comentar-se-á a sua versão atual, a 0.4.0.

2. BLUETOOTH

Em 1994, a empresa Ericsson Mobile Communication – encabeçada por Jaap Haarsen – iniciou, em Lund, na Suécia, pesquisas sobre a possibilidade do desenvolvimento de uma tecnologia que permitisse a comunicação sem fio entre seus dispositivos móveis e acessórios, que apresentasse um baixo consumo de energia. Esse estudo também era parte de um projeto maior, que investigava a possibilidade de multi-comunicadores estarem conectados à rede celular via telefones móveis. Desse modo surgiu o Bluetooth e, como muitos disseram na época, iria substituir outras tecnologias já bastante difundidas no mercado como o infravermelho, as conexões via interfaces USB e outros padrões que utilizavam cabo. Entretanto, sabe-se, como acontece com toda e qualquer tecnologia, que existem casos nos quais uma determinada solução é melhor e em outros nem tanto. Por isso, todas essas tecnologias ainda estão trabalhando em conjunto e desempenhando papéis de importâncias distintas.

No começo de 1997, quando os projetistas da Ericsson já estavam trabalhando no *microchip* transmissor/receptor do Bluetooth, a empresa aproximou-se de outros fabricantes de dispositivos portáteis tentando promover o uso desta tecnologia. Tal estratégia foi adotada porque, para o Bluetooth ser um sucesso, seria necessário que uma massa crítica de aparelhos portáteis usassem esse mesmo padrão para comunicação. Então, a Ericsson, mostrou-se interessada em que outras empresas também trabalhassem no desenvolvimento da tecnologia. Sendo assim, em fevereiro de 1998, foi criado o *Special Interest Group* (SIG) do Bluetooth em parceria com a Nokia, IBM, Toshiba e Intel, reunindo, assim, empresas referentes às áreas de telefonia móvel, *notebooks* e processadores. O consórcio foi anunciado para o mundo em maio daquele mesmo ano e, no decorrer dos tempos, teve um grande crescimento contando com a adesão de outras renomadas empresas como a Motorola, Dell, 3Com, Compaq, Qualcomm., Samsung, Siemens, Symbian, entre outras. Atualmente o SIG do Bluetooth já conta com mais de 8000 empresas participantes (BLUETOOTH SIG, 2006).

2.1 Origem do Nome e Logo

O nome Bluetooth surgiu em homenagem a um rei viking do século X chamado de Danish King Harald Blatand - ou Harald Bluetooth, em inglês (BLUETOOTH MEMBERSHIP, 2006). Esse rei unificou partes da Noruega, Suécia e Dinamarca. Durante o estágio de formação do SIG, optou-se por esse codinome, que depois tornou-se realmente o nome, pelo fato da tecnologia também querer representar essa união através das indústrias distintas de computação, telefonia celular e mercados automotivos.

O logo do Bluetooth, mostrado na Figura 2.1, foi criado por uma empresa da Escandinávia. Ele combina a letra “B” com a letra “H”, em um alfabeto antigo, que aparentemente é muito similar a um asterisco.

Figura 2.1 – Logo Bluetooth (BLUETOOTH SIG, 2006).

2.2 Infra-Estrutura

Bluetooth é uma tecnologia sem fio robusta, de baixo consumo e custo que tem como intuito interconectar diferentes dispositivos. Ele utiliza sinais de radiofrequência para isso.

A banda de radiofrequência utilizada pelo Bluetooth opera na faixa *Industrial, Scientific, Medical* (ISM) - que é a banda utilizada por instrumentos industriais, científicos e médicos - centrada em 2,45 GHz. Entretanto, nos Estados Unidos, a faixa ISM varia de 2,4GHz a 2,4835

GHz, enquanto que no Japão, de 2,4 GHz a 2,5 GHz. Para a operação de dispositivos Bluetooth em países como Espanha e França são necessários alguns ajustes, pois a largura de banda e a localização da faixa ISM também diferem. Entretanto, já existem iniciativas para que o espectro de freqüência da faixa ISM esteja globalmente disponível com objetivo de assegurar uma compatibilidade mundial das comunicações.

A comunicação entre os dispositivos Bluetooth é feita através do estabelecimento de um canal *Frequency Hopping - Code-Division Multiple Access* (FH-CDMA). Nesta técnica, o transmissor envia um sinal sobre uma série aparentemente randômica de freqüências de rádio. Um receptor, sintonizando entre tais freqüências, em sincronia com o transmissor, capta o sinal. A mensagem é totalmente recebida apenas se o receptor conhecer essa série de freqüências que o transmissor usará para enviar o sinal.

O Bluetooth divide a banda passante entre 79 portadoras espaçadas de 1 MHz. Portanto, existem 79 freqüências nas quais instantaneamente um dispositivo pode estar transmitindo. A seqüência particular de freqüências de um canal é estabelecida pelo dispositivo mestre, que é o responsável pelo controle do canal. Todos os outros dispositivos participantes são nomeados escravos e devem sincronizar ao mestre. O dispositivo mestre muda sua freqüência de transmissão 1600 vezes por segundo, com o objetivo de minimizar potenciais interferências de dispositivos terceiros.

O Bluetooth pode ser dividido em três classes: 1, 2 e 3. A primeira é para equipamentos com alcance de aproximadamente 100 metros, a segunda, 10 metros e a terceira, 1 metro.

2.3 Piconet e Scatternet

Uma rede *piconet* é a forma usual de uma rede Bluetooth e consiste em um dispositivo mestre (*master*) e um ou mais escravos (*slaves*) conectados a ele. O dispositivo que inicia uma conexão Bluetooth se torna o mestre da rede. Uma *piconet* pode possuir até sete escravos conectados a um mestre. Os escravos podem transmitir dados apenas quando o mestre define esse tempo de transmissão a eles. Além disso, escravos não podem se comunicar diretamente

entre eles, fazendo com que toda comunicação tenha que passar primeiramente pelo mestre. Os escravos sincronizam sua faixa de freqüência baseados no mestre e no seu endereço Bluetooth.

Piconets tem uma forma de rede estrela, com o mestre no nodo central, conforme ilustrado na Figura 2.2. Duas ou mais piconets podem se interconectar, fazendo com que se tornem uma *scatternet*, como mostra a Figura 2.3. Essa conexão é feita por um nodo intermediário, que compartilha o tempo seguindo a faixa de freqüência de cada *piconet* por vez. Isso reduz a fatia de tempo disponível para transferência de dados entre o nodo intermediário e o mestre, dinvindo-a, pelo menos, pela metade.

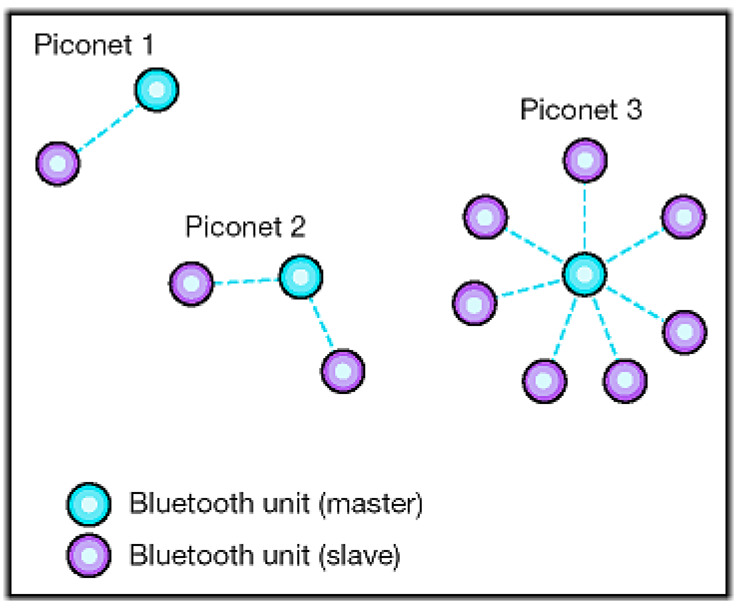


Figura 2.2 – *Piconet* (FRODIGH, 2002).

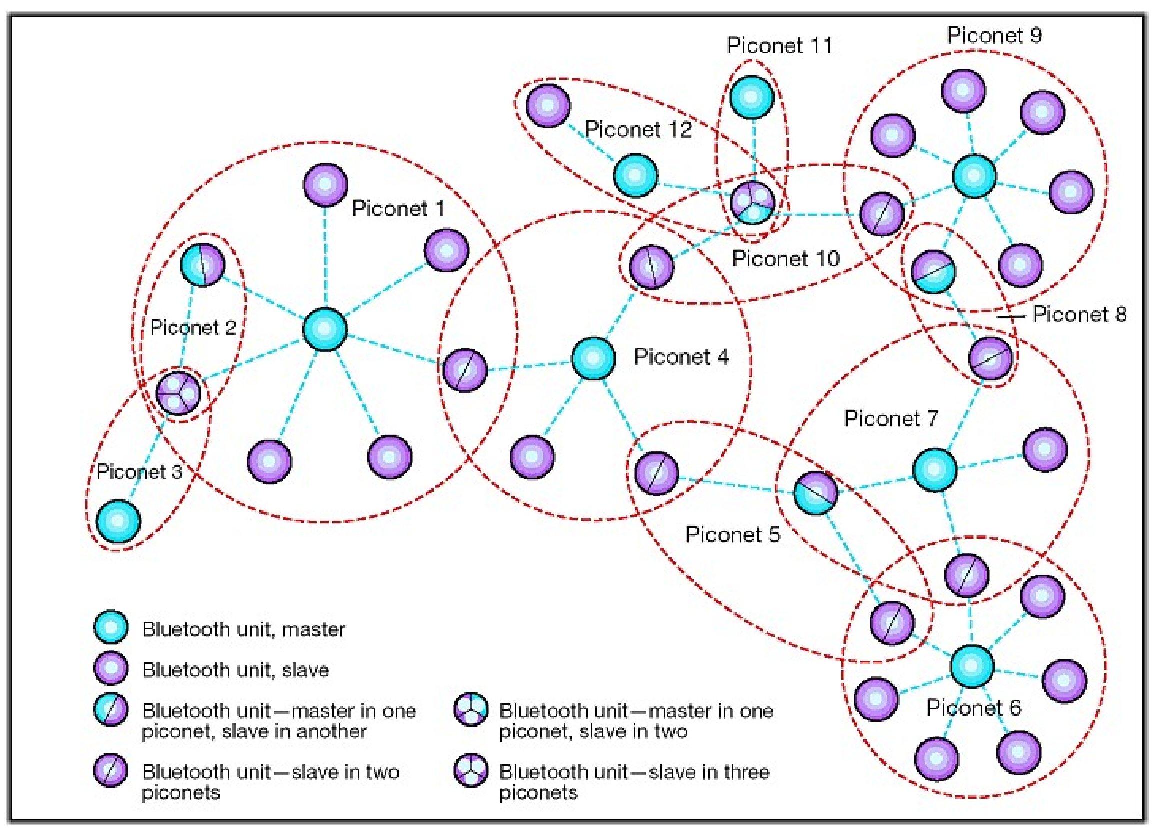


Figura 2.3 – *Scatternet* (FRODIGH, 2002).

2.4 Pesquisa por Dispositivos e Serviços

Devido à natureza das redes Bluetooth, dispositivos remotos entram e saem da rede devido ao seu espaço de abrangência. Os dispositivos Bluetooth possuem a habilidade de descobrir outros dispositivos que estejam próximos e descobrir também quais os serviços que esses dispositivos podem oferecer (KLINGSHEIM , 2004).

Durante o processo de pesquisa por dispositivo, o dispositivo que está pesquisando receberá endereços Bluetooth juntamente com as frequências dos dispositivos encontrados. Assim sendo, esse dispositivo saberá identificar todos os dispositivos encontrados pelos seus endereços Bluetooth.

Dispositivos tornam-se possíveis de serem encontrados quando entram no modo *inquiry scan*.

Nesse modo, o dispositivo irá gastar um período de tempo mais longo que o normal em cada canal. Isso aumenta a possibilidade de detecção de dispositivos que estão pesquisando. Além disso, os dispositivos descobertos fazem uso do *Inquiry Access Code* (IAC). Existem dois tipos de IAC, o *General Inquiry Access Code* (GIAC) e o *Limited Inquiry Access Code* (LIAC). O primeiro é usado quando a pesquisa é feita por um período indefinido de tempo e o segundo, por sua vez, quando há um período limitado de tempo.

2.5 Serviços Bluetooth

Diferentes dispositivos Bluetooth podem oferecer diferentes serviços. Um dispositivo Bluetooth, após ter encontrado um outro dispositivo remoto, poderá fazer uma pesquisa pelo serviços disponíveis nesse dispositivo remoto ou apenas por algum serviço específico. O serviço de pesquisa de dispositivos utiliza o protocolo SDP (*Service Discovery Protocol*). Em um contexto SDP, um cliente envia requisições a um servidor que, por sua vez, envia as informações de serviços disponíveis nele.

Dispositivos Bluetooth mantém informações sobre seus serviços em um *Service Discovery DataBase* (SDDB). Um SDDB contém entradas de registros de serviços onde cada um desses registros contém atributos que o descrevem, conforme ilustrado na Figura 2.4.

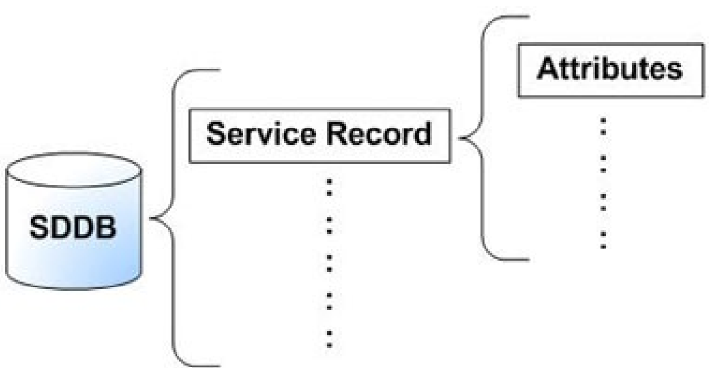


Figura 2.4 – *Service Discovery DataBase* (ORTIZ, 2006).

Cada atributo é representado por um número hexadecimal chamado de identificador do atributo. Apesar de um registro poder ter vários atributos, dois sempre obrigatoriamente deverão existir: o *ServiceRecordHandle* (identificador de atributo 0x0000) e *ServiceClassIDList* (identificador de atributo 0x0001). A Tabela 2.1 mostra alguns identificadores de atributos comumente utilizados.

Attribute Name	Attribute ID	Attribute Value Type
ServiceRecordHandle	0x0000	inteiro de 32 *bits* sem sinal
ServiceClassIDList	0x0001	DATASEQ de UUIDs
ServiceRecordState	0x0002	inteiro de 32 *bits* sem sinal
ServiceID	0x0003	UUID
ProtocolDescriptorList	0x0004	DATASEQ de DATASEQ de UUIDs e parâmetros opcionais
BrowseGroupList	0x0005	DATASEQ de UUIDs
LanguageBasedAttributeIDList	0x0006	DATASEQ de triplas de DATASEQ
ServiceInfoTimeToLive	0x0007	inteiro de 32 *bits* sem sinal
ServiceAvaliability	0x0008	inteiro de 8 *bits* sem sinal
BluetoothProfileDescriptoList	0x0009	DATASEQ de pares de DATASEQ
DocumentationURL	0x000A	URL
ClientExecutableURL	0x000B	URL
IconURL	0x000C	URL
VersionNumberList	0x0200	DATASEQ de inteiros de 16 *bits* sem sinal
ServiceDatabaseState	0x0201	inteiro de 32 *bits* sem sinal

Tabela 2.1 – Identificadores dos serviços mais comuns e seus tipos (adaptado de BLUETOOTH MEMBERSHIP, 2006).

Atributos diferentes contém valores de vários tipos e tamanhos, que são armazenados por um elemento de dados. Um elemento de dados consiste em duas partes: a primeira é um descritor do tipo do elemento e a segunda é um campo de dados. O descritor do tipo do elemento contém informações sobre o tipo e o tamanho do dado, enquanto que o campo de dados contém o dado em si. Sendo assim, um dispositivo remoto sempre saberá qual o tipo de dado

e qual seu tamanho quando estiver pesquisando um determinado atributo.

O *Universally Unique Identifier* (UUID) é o tipo de dado usado para identificar os serviços, protocolos, perfis, etc. Um UUID é um identificador de 128 *bits* que garante ser único em um tempo e espaço determinado. A tecnologia Bluetooth utiliza diferentes variantes de UUIDs, com UUID pequenos e grandes. Isso é feito com o intuito de reduzir eventuais armazenamento e transferências desnecessárias de valores de UUIDs de 128 *bits*. Existe uma gama de UUIDs pequenos pré-alocados para serviços frequentemente usados, protocolos e perfis.

2.6 Arquitetura

O intuito da especificação Bluetooth é permitir que dispositivos de fabricantes distintos interconectem-se de forma compatível e interoperável. Por conta disso, não é apenas suficiente a existência do sistema de rádio em *hardware*, havendo também a necessidade de uma complexa pilha de protocolos em *software* que garantam todo esse funcionamento (BLUETOOTH MEMBERSHIP, 2006).

A pilha de protocolos Bluetooth é dividida em duas porções de camadas: inferior e superior. As camadas que compõem a pilha de protocolos são ilustradas na Figura 2.5. Fazem parte da porção inferior a camada de rádio, banda de base, controlador de enlace e gerenciador de enlace, enquanto que na porção superior estão contidas a L2CAP, RFCOMM, OBEX, e os perfis.

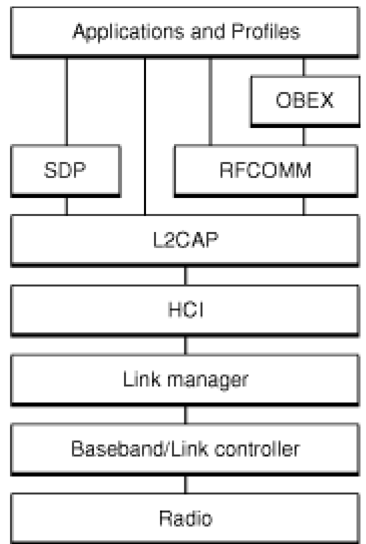


Figura 2.5 – Pilha de Protocolos Bluetooth (APPLE, 2006b).

2.6.1 Camadas Inferiores

Na base da pilha está a camada de rádio. O módulo de rádio em um dispositivo Bluetooth é responsável por modular e demodular os dados em freqüências de sinais de rádio para transmissão e captação no ar. A camada de rádio descreve quais as características físicas que o componente transmissor e receptor do dispositivo Bluetooth deve conter. Dentre essas, incluem-se as características da modulação, tolerância à freqüência e o grau de sensibilidade.

Acima da camada de rádio está a camada de banda de base (*baseband*) juntamente com o controlador de enlace (*link controller*). A especificação Bluetooth não estabelece uma clara distinção entre as responsabilidades das duas. Entretanto, é conveniente dizer que a camada de banda de base é responsável por formatar propriamente os dados para transmissão recebidos e enviados para a camada de rádio. Além disso, ela manipula a sincronização da comunicação. Já o controlador de enlace é responsável por estabelecer e manter a comunicação com a camada acima.

O gerenciador de enlace (l*ink manager*) traduz os comandos da interface de controle do *host* (*host controller interface*, também chamada de HCI) que ele recebe em operações na camada

de banda de base. Ele é responsável por estabelecer e configurar as comunicações e gerenciar as requisições, entre outras tarefas.

A camada HCI atua como uma interface entre as camadas inferiores e superiores da pilha de protocolos Bluetooth. A especificação Bluetooth define uma camada HCI padrão para suportar sistemas Bluetooth que são implementados sobre dois processos diferentes: possuindo dois processadores distintos ou apenas um único. Por exemplo, um sistema Bluetooth em um computador pode usar um módulo de processador para implementar as camadas inferiores da pilha e um processador próprio para implementar as camadas superiores. Nesse conceito, a porção inferior é chamada de módulo Bluetooth e a superior de *host* Bluetooth. Entretanto, não é necessário dividir a pilha Bluetooth dessa maneira, pois alguns dispositivos, por exemplo os fones de ouvido sem fio, combinam esse módulo e a porção do *host* da pilha em apenas um processador, devido à necessidade de serem pequenos e condensados. Além disso, pelo fato de que a HCI é bem arquitetada, é possível implementar *drivers* que manipulem diferentes módulos Bluetooth de diversos fabricantes.

2.6.2 Canais de Comunicação

A especificação Bluetooth define dois tipos de comunicação entre dispositivos Bluetooth:

- *Synchronous Connection-Oriented Link* (SCO) : É a conexão para transmitir áudio. Funciona do tipo ponto-a-ponto, simétrico onde as parcelas de tempo ficam reservadas e a rede funciona como se fosse por comutação de circuito. Este tipo de configuração atinge 64Kbps, taxa exigida para a transmissão de dados de voz PCM, e é ideal para a comunicação de pacotes de voz. Os pacotes SCO podem ser enviados com redundância para corrigir a perda de pacotes visto que não há reenvio da informação.
- *Asynchronous Connection-Less Link* (ACL): É a conexão para transmitir dados. Funciona do tipo ponto-a-multiponto e garante a retransmissão de pacotes. Este tipo de conexão é análoga à rede de comutação de pacotes e passa a existir quando é feita uma conexão entre um dispositivo mestre e um escravo.

2.6.3 Camadas Superiores

Acima da camada HCI está a porção superior da pilha de protocolos Bluetooth. A primeira camada delas é a *Logical Link Control and Adaptation Protocol* (L2CAP). Ela é responsável por estabelecer conexões entre os enlances ACL existentes, ou requisitar um se ele ainda não existir; pela multiplexação entre os diferentes protocolos das camadas acima, tais como o *RS-232 Serial Cable Emulation Profile* (RFCOMM) e o *Service Discovery Protocol* (SDP), no intuito de permitir que muitas aplicações diferentes usem um enlace ACL; e por reempacotar os pacotes de dados que receber das camadas acima no formato esperado pelas camadas abaixo.

A L2CAP aborda o conceito de canais para a informação da onde os pacotes vieram e para onde estão indo. Esse canal seria uma representação lógica do fluxo de dados entre a camada L2CAP nos dispositivos remotos.

Acima da camada L2CAP não existe uma ordem exigida das próximas camadas por elas se tratarem de camadas compostas. Irá ser descrito primeiramente a camada SDP.

A SDP é comum em todos dispositivos e define ações para servidores e clientes de serviços Bluetooth. A especificação define um serviço como qualquer coisa que é usada remotamente por outro dispositivo Bluetooth. Um dispositivo pode ser, ao mesmo tempo, um servidor e cliente.

Um cliente SDP se comunica com um servidor SDP usando um canal reservado, sob o enlace L2CAP, para pesquisar quais serviços estão disponíveis. Quando um cliente encontra um serviço desejado, ele requisita uma conexão separada para usar esse serviço. O canal reservado é dedicado para comunicação SDP, então um dispositivo sempre sabe como conectar-se ao serviço SDP em qualquer dispositivo, ou seja, um dispositivo sempre sabe como fazer a pesquisa por serviços. Um servidor SDP mantém o SDDB.

Também acima da camada L2CAP está a camada RFCOMM. O protocolo RFCOMM emula uma configuração de cabo serial em uma porta RS-232. O RFCOMM se conecta às camadas

inferiores da pilha de protocolos Bluetooth através da camada L2CAP. Provendo essa emulação da porta serial, o RFCOMM suporta diferentes aplicações legadas que façam uso da porta serial, além de também permitir implementações de protocolos mais complexos sobre ele, como o *Object Exchange* (OBEX) e os perfis Bluetooth.

OBEX é um protocolo de transferência que define dados de objetos e um protocolo de comunicação para que dois dispositivos possam usar para facilmente trocar esse objetos (vCard, vCalendar, imagens, vídeos, etc). O OBEX tem origem na especificação do infravermelho e o Bluetooth adotou-o devido ao fato das camadas inferiores desse primeiro protocolo, o *Infrared* OBEX (IrOBEX), serem muito similares às da camadas de sua pilha de protocolos. Além disso, o IrOBEX é amplamente aceito e usado, o que também foi, de fato, uma ótima escolha realizada pelo SIG para a promoção do Bluetooth, através de tecnologias pré-existentes.

2.7 Perfis Bluetooth

A especificação Bluetooth define uma gama de perfis para diferentes tipos de tarefas, sendo que alguns desses ainda nem se quer foram implementados por qualquer dispositivo ou sistema. Os perfis garantem a interoperabilidade entre dispositivos de fabricantes distintos. Sendo assim, consumidores podem comprar um celular de um fabricante e um fone de ouvido sem fio de outro, por exemplo, e os dois funcionarão adequadamente, assumindo que os dois possuem suporte a esse perfil necessário.

Os perfis possuem três características padrão:

- Dependência de outros perfis: Todo perfil depende do perfil de base, chamado de perfil de acesso genérico (*Generic Access Profile*, comumente chamado de GAP), e alguns outros dependem também de perfis intermediários.
- Formato padrão para interface com usuário: Cada perfil descreve como um usuário deve ver o perfil, fazendo com que seja mantida uma experiência consistente a ele.
- Especificação de partes da pilha de protocolos Bluetooth usadas pelo perfil: Cada perfil utiliza opções e parâmetros distintos para cada camada da pilha.

A Figura 2.6 representa a estrutura de alguns perfis mais comuns e suas descrições. Mais informações sobre os novos perfis que estão surgindo podem ser obtidas em BLUETOOTH MEMBERSHIP (2006).

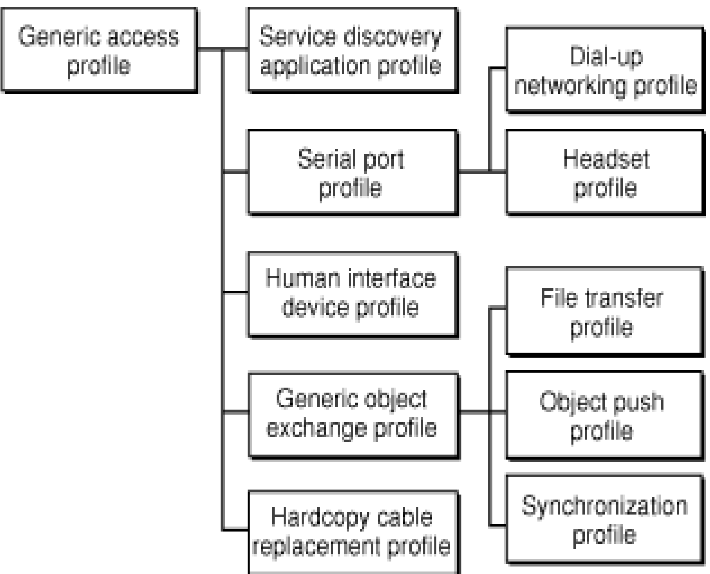


Figura 2.6 – Perfis Bluetooth (APPLE, 2006b).

Na base da hierarquia dos perfis está o GAP. Ele é responsável por definir uma forma consistente de estabelecer a conexão entre os dispositivos Bluetooth. Além disso, o GAP também define quais funcionalidades devem ser implementadas nos dispositivos, procedimentos genéricos para descobrir e conectar dispositivos e a terminologia básica para interface com o usuário.

O perfil *Service Discovery Application* (SDA) permite que uma aplicação use SDP para pesquisar serviços ou dispositivos remotos. Como requerido pelo GAP, um dispositivo Bluetooth deve poder se conectar a um outro. Baseado nisso, o SDP necessita que cada aplicação consiga pesquisar quais serviços estão disponíveis em qualquer dispositivo Bluetooth e se conectar a eles.

O perfil *Human Interface Device* (HID) permite a comunicação com uma classe de dispositivo HID usando conexões Bluetooth. Ele fornece meios de usar o protocolo USB HID para pesquisar funcionalidades sobre classes de dispositivos HID e também prevê como um dispositivo Bluetooth pode suportar serviços HID usando a camada L2CAP.

O *Serial Port Profile* (SPP), como o nome sugere, suporta a emulação de uma configuração de cabo serial em uma porta RS-232 para dispositivos Bluetooth. Sendo assim, esse perfil

permite que, sem nenhuma modificação, aplicações legadas possam utiliza o Bluetooth como se o mesmo fosse uma conexão a uma porta serial. O SPP utiliza o protocolo RFCOMM para prover essa emulação.

O perfil *Dial-Up Networking* (DUN) roda sobre o SPP e descreve como um dispositivo de terminal de dados, como um *laptop*, pode usar um outro dispositivo para servir de *gateway*, como um celular ou modem, para acessar uma rede baseada em um telefone. Como outros dispositivos que rodam sobre o SPP, a conexão virtual serial criada pelas camadas inferiores da pilha de protocolos Bluetooth é transparente para as aplicações usando o perfil DUN. Além disso, o *driver* do modem no dispositivo de terminal de dados não sabe que está se comunicando por Bluetooth, nem tampouco a aplicação nele sabe que não está conectada por um cabo.

O *Headset Profile* (HSP) descreve como um fone de ouvido deve se comunicar com um computador ou outro dispositivo Bluetooth, tal como um telefone celular. Quando conectado e configurado, o fone de ouvido atua como uma interface de entrada e saída remota de áudio.

O *Hardcopy Cable Replacement Profile* (HRCP) descreve como enviar dados renderizados sobre uma conexão Bluetooth com outro dispositivo, como uma impressora. Mesmo que outros perfis possam ser usados para impressão, o HRCP é especialmente projetado para suportar aplicações de impressão.

O *Generic Object Exchange Profil* (GOEP) provê um *blueprint* genérico para outros perfis usando o protocolo OBEX e define regras para dispositivos clientes e servidores. Tal como todas as outras transações OBEX, esse perfil estipula que o cliente inicie toda a transação. Entretanto, esse perfil não descreve como aplicações devem definir os objetos para troca ou como as aplicações devem implementar essa troca. Esses detalhes são responsabilidades dos perfis que dependem dele: o *Object Push Profile* (OPP), *File Transfer Profile* (FTP) e *Syncronization Profile* (SYNC).

O OPP define regras para clientes e servidores de leitura. Essas regras são análogas e devem ser interoperáveis entre dispositivos servidores e clientes distintos, baseados nas regras definidas pelo GOEP. O OPP retém-se a uma gama de formatos de objetos para uma máxima interoperabilidade. O mais comum e aceito deles é o formato vCard, que é o formato para representar um cartão de negócios (usado na agenda de contatos do celular). Se uma aplicação necessita trocar dados em outros formatos, ela deve usar outro perfil como o FTP.

O FTP provê uma guia de regras para aplicações que necessitam trocar objetos, tais como

pastas e arquivos, o que oferece uma gama muito maior de objetos que apenas os suportados pelo OPP. O FTP também descreve regras para clientes e servidores e uma gama de outras possibilidades em cenários distintos. Por exemplo, se um cliente desejar pesquisar os objetos disponíveis no servidor, é necessário que ele suporte a funcionalidade de *pull*. Assim, também é necessário que o servidor requerido responda a essa requisição provendo a resposta correta.

O SYNC descreve como as aplicações devem realizar a sincronização dos dados entre dispositivos, tais como entre um PDA e um computador. Ele também define regras para clientes e servidores. O SYNC atém seu foco no gerenciamento da troca de dados de informações pessoais, tais como os de um calendário (vCalendar). Esse pefil também define como uma aplicação pode suportar sincronização automática de dados – sincronização que ocorre quando os dispositivos pesquisam e se conectam sem algum comando externo do usuário.

2.8 Segurança

Segurança é algo preocupante no mundo em que se vive hoje e é também alvo de muitos estudos quando se fala em tecnologias de redes sem fio. A todo lugar há dados privados sendo trafegados entre dispositivos interligados via rede.

Em redes Bluetooth, por exemplo, os dispositivos habilitados são passíveis de serem encontrados por qualquer pessoa em um raio determinado. Sendo assim, qualquer um poderia tentar se conectar aos serviços oferecidos em algum dispositivo alvo ou então utilizar-se de técnicas de intrusão para roubo de informações. Para lidar com esses problemas, a especificação Bluetooth definiu um modelo de segurança baseado em três componentes: autenticação, autorização e encriptação. Além disso, três modos de segurança também foram definidos para garantir diferentes níveis de segurança entre o modelo, juntamente com o gerenciador de segurança criado com objetivo de tratar as transações que envolvam esses aspectos.

2.8.1 Modos de Segurança

Os modos de segurança são parte do perfil GAP, que é de implementação obrigatória para dispositivos Bluetooth qualificados. O fabricante é quem deve decidir qual modo de segurança que será suportado quando implementar o perfil GAP no dispositivo Bluetooth. Além disso, por exemplo, em dispositivos com maior poder de processamento, como *notebooks*, o fabricante poderia permitir que o usuário escolha qual modo usar para o mesmo poder se adequar a diversas situações. Os modos de segurança que o perfil GAP define são:

1. Sem segurança
2. Segurança em nível de serviço
3. Segurança em nível de conexão

No modo de segurança 1 os dispositivos nunca vão iniciar nenhum procedimento de segurança e o suporte a autenticação é opcional. Esse modo não é muito comum, pois a maioria dos dispositivos necessitam de algum suporte seguro.

Já o modo de segurança 2 é usado pela maioria dos dispositivos. Esse modo define que a segurança é garantida em nível de serviço, ou seja, é o serviço quem decide se ela será ou não necessária. Os procedimentos do modo de segurança 2 são iniciados pelas camadas superiores da pilha de protocolo Bluetooth, o que permite que desenvolvedores decidam para qual serviço será necessário suporte a segurança.

Além disso, no modo 2, dispositivos e serviços também possuem diferentes níveis de segurança. Para dispositivos, há ainda dois níveis: dispositivos pareados, dispositivos conhecidos que não foram pareados e dispositivos desconhecidos. Um dispositivo pareado – ou seja, que já foi conectado com o outro dispositivo em questão - tem acesso irrestrito a todos os serviços. Quanto aos serviços, há três níveis definidos: serviços que requerem autorização e autenticação, serviços que requerem apenas autenticação e serviços abertos a todos dispositivos.

No modo de segurança 3 os procedimentos de segurança são iniciados durante a configuração

da conexão Bluetooth. Isso quer dizer que se algo falhar, a conexão falhará. Esse modo é iniciado pelas camadas inferiores e desenvolvedores não podem configurá-lo. Ele é muito utilizado em dispositivos com configurações prévias que não podem sofrer alterações, como por exemplo, os fones de ouvido sem fio.

2.8.2 Autenticação (Pareamento e Ligação)

Ligação (*Bonding*) é o procedimento de autenticação de um dispositivo Bluetooth realizado por outro dispositivo. Esse procedimento é dependente de uma chave de autenticação comum (também chamada de chave de combinação). Se os dispositivos não compartilham essa chave, ela será criada antes do processo de ligação ser finalizado. Essa geração da chave de autenticação é chamada de pareamento (*pairing*).

O procedimento de pareamento envolve geração de uma chave de inicialização e autenticação, seguido de uma autenticação mútua. A chave de inicialização é baseada em uma entrada do usuário, um número randômico e o endereço Bluetooth de um dos dispositivos. Essa entrada do usuário é chamada de *Personal Identification Number* (PIN) ou chave de acesso e deve ter até 128 *bits* de tamanho. Essa chave de acesso é trocada em segredo entre os dois dispositivos. A chave de autenticação é baseada em números randômicos combinados com o endereço dos dois dispositivos Bluetooth. A chave de inicialização é usada para encriptar dados quando trocados para criar a chave de autenticação e depois é descartada. Quando o processo de pareamento é finalizado, os dispositivos estarão dividindo essa mesma chave de autenticação e ter-se-ão autenticado um ao outro.

Quando dois dispositivos completam o processo de pareamento, eles podem guardar a chave de autenticação para uso posterior. Após a chave ser salva, é possível conectar esses dispositivos automaticamente em outros momentos sem a necessidade do uso da chave de inicialização. Isso é muito útil em sistemas que usam de conexões freqüentes, como no caso de um PDA que todo dia sincroniza uma agenda com um computador.

2.8.3 Autorização

Autorização é o processo de dar permissão de acesso a um dispositivo Bluetooth remoto que esteja acessando um serviço. Antes de um dispositivo ser autorizado, é necessário que ele primeiro seja autenticado. As permissões de acesso podem ser dadas temporariamente ou de forma permanente. Sendo assim, um dispositivo que já foi pareado com outro pode ter acesso permanente – até que alguém queira excluir isso - a determinados serviços ou então, ele poderia ter acesso apenas em um dado momento para, por exemplo, trocar algum dado, como uma imagem.

2.8.4 Encriptação

Quando dois dispositivos se autenticaram, a encriptação pode ser requerida pela conexão Bluetooth. Antes da encriptação começar, é necessário que os dispositivos negociem o modo que ela se dará e o tamanho da chave de encriptação. Existem três modos de encriptação:

- sem encriptação
- ecriptação de pacotes ponto-a-ponto e pacotes broadcast
- encriptação apenas de pacotes ponto-a-ponto

Quando apenas dois dispositivos estão conectados à rede, a encriptação ponto a ponto de pacotes é a escolha natural nessa cado. Já o modo sem encriptação será usado se um dos dois dispositivos não suportar encriptação.

Quando a encriptação for requisitada e os dois dispositivos a suportarem, o tamanho da chave de encriptação será negociada. O dispositivo mestre irá sugerir o tamanho máximo de chave

que ele suporta. O dispositivo escravo poderá aceitá-lo ou não. Se ele aceitar o tamanho de chave sugerido pelo mestre, a encriptação pode começar. Se ele o rejeitar, o mestre pode sugerir uma chave menor ou decidir por terminar a conexão. Esse procedimento é repetido até os dispositivos concordem com um tamanho ou o mestre decidir por terminar a conexão. Os tamanhos de chave de 8 a 128 *bits* são suportados para serem chaves de encriptação.

2.8.5 Gerenciador de Segurança

As informações dos dispositivos pareados e dos diferentes níveis de autorização para os diferentes serviços são armazenadas em dois bancos de dados distintos, um para cada uma dessas funções. Como muitas camadas necessitam acessar esses bancos de dados, é o gerenciador de segurança quem permite acesso uniforme a eles, adicionando e extraindo informações. Além disso, aplicações que usem funcionalidades envolvendo segurança precisam registrá-las com o gerenciador de segurança.

O gerenciador de segurança também é responsável por pedir ao usuário uma chave de acesso durante o processo de pareamento e uma resposta confirmando a autorização quando um dispositivo remoto tenta conectar a um serviço que necessite de autorização.

2.8.6 Ataques a Dispositivos

O SIG do Bluetooth investiga os problemas de segurança reportados para tentar descobrir as suas causas. Quando a causa está na própria tecnologia Bluetooth, é feita a atualização da especificação para corrigi-la. Entretanto, muitas vezes, a causa está no fabricante, que possui algum dispositivo com uma implementação errônea ou com algum outro tipo de vulnerabilidade na plataforma.

Para coibir esses problemas, também é necessário que os fabricantes implementem algum

nível mínimo de segurança disponível no perfil GAP. A seguir são descritos alguns problemas conhecidos. A maioria deles são antigos e, por serem tão conhecidos e problemáticos, já foram devidamente corrigidos:

- *Bluejacking* – Permite que o usuário do telefone envie cartões de contatos (vCard) anonimamente usando a tecnologia Bluetooth. O *Bluejacking* não remove ou altera qualquer dado no dispositivo alvo, ele apenas envia esses cartões com intuito de fazer com que usuários leigos não saibam corretamente o que está acontecendo e acabem por aceitar esse cartões.
- *Bluebugging* – Permite que indivíduos habilidosos explorem falhas em telefones celulares usando a tecnologia Bluetooth, sem que o usuário alvo receba qualquer alerta ou notificação pelo dispositivo. Essas falhas permitem que estes indivíduos acessem comandos do celular e possam, assim, iniciar ligações, enviar e receber mensagens de texto, ler e escrever na agenda de contatos, conectar à Internet, etc.
- *Bluesnarfing* – Permite que indivíduos ganhem acesso a dados salvos no telefone alvo através da tecnologia Bluetooth, sem que o possuidor desse telefone aceite qualquer conexão ou receba qualquer alerta. Os dados incluem agenda de contatos, imagens, calendário, etc.
- *Car Whisperer* – Foi um *software* desenvolvido por alguns especialistas de segurança com intuito de mostrar vulnerabilidades em implementações de alguns kits Bluetooth disponíveis em carros. O *software* permitia se conectar e comunicar com um carro sem estar autorizado através de um dispositivo remoto e assim, enviar áudios para as caixas de som e receber áudios do veículo.
- *Cabir Worm* – É um *software* malicioso - um *malware* - que quando instalado em algum telefone, envia cópias de si mesmo para outros dispositivos através do Bluetooth. Ele só pode ser instalado em alguns tipos específicos de sistemas operacionais de celulares e também, para sua propagação, o dono do outro dispositivo deve aceitá-lo quando ele tentar se instalar.
- *Denial of Service* (DoS) *Attack* – Esse tipo de ataque se consagrou na Internet e agora também é uma opção para dispositivos com o Bluetooth ativado. Consiste em fazer requisições seguidas no intuito de consumir a carga da bateria do dispositivo e bloquear possíveis requisições que venham a ser feitas para algum serviço do dispositivo.

3 JAVA

A tecnologia Java foi desenvolvida na década de 90 pelo programador James Gosling, que pertencia à empresa Sun Microsystems. O principal componente dessa tecnologia é uma linguagem de programação que ficou conhecida pelos desenvolvedores apenas por Java. Diferentemente das linguagens convencionais que são compiladas para código nativo, a linguagem Java é compilada para um código intermediário - chamado de *bytecode* - que é executado por uma máquina virtual, a Java Virtual Machine (JVM). Essa facilidade do código ser interpretado em tempo de execução faz com que a linguagem seja independente de plataforma, consagrando uma das maiores características da mesma: a portabilidade.

Devido a uma séria de fatores, incluindo o aumento do seu uso e a disponibilidade de bibliotecas padrão da liguagem, ela foi dividida em três plataformas: *Java Platform, Standard Edition* (Java SE); *Java Platform, Enterprise Edition* (Java EE); e *Java Platform, Micro Edition* (Java ME).

O Java SE é projetado para execução em máquinas simples de computadores pessoais a estações de trabalho, comumente utilizada para o aprendizado da linguagem. O Java EE é destinado a aplicativos baseados no servidor, e possui uma séria de pacotes adicionais, necessários para seu contexto de uso, e não será abordada nesse trabalho. Por fim, o Java ME é destinado a dispositivos com memória, vídeo e processamento limitado, o que torna a linguagem uma derivação reduzida – mas diferente - do Java SE.

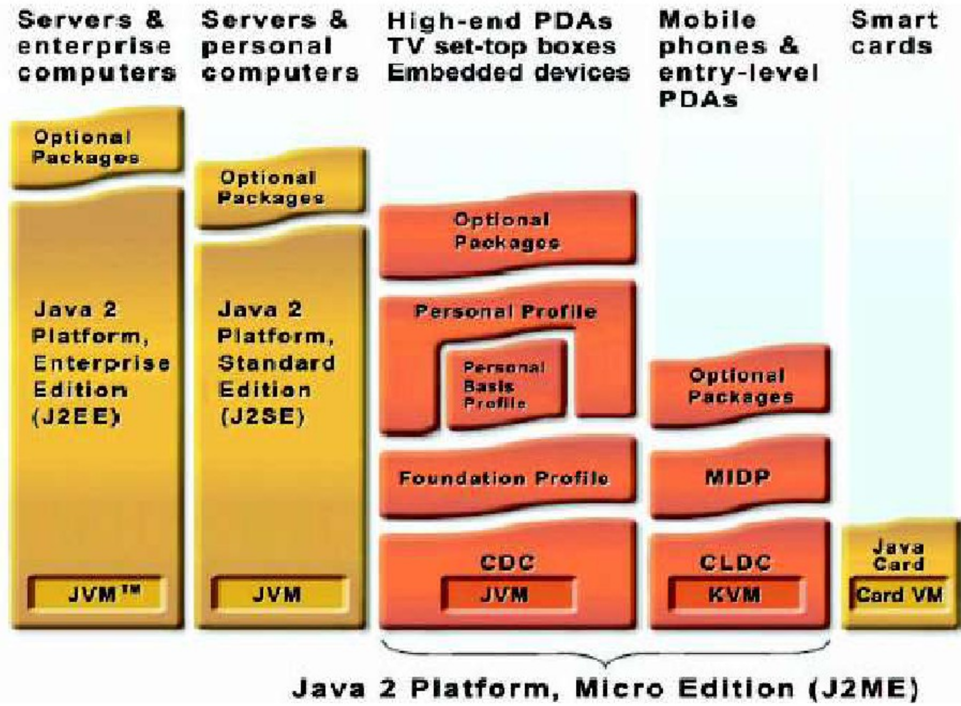


Figura 3.1 - Plataforma Java (SUN, 2006b).

3.1 Java Plataform, Standard Edition (Java SE)

O Java SE é a plataforma para desenvolvimento Java de aplicações para desktop e servidores, usada como base para as demais plataformas [SUN MICROSYSTEMS, 2006b]. Ele possui uma arquitetura robusta, conforme ilustrado na Figura 3.2, e está dividido em três partes:

- Componentes do núcleo ou *core*: Provê funcionalidades essenciais para escrita de programas, para acesso a banco de dados, segurança, invocações de procedimentos remotos e comunicação.
- Componentes do cliente ou *desktop*: Provê facilidades para ajudar na construção de aplicações para os usuários. Isso inclui componentes de desenvolvimento como os Java Plug-in, APIs de modelos de componentes como os JavaBeans, interfaces gráficas para os usuários, etc.
- Outros componentes.

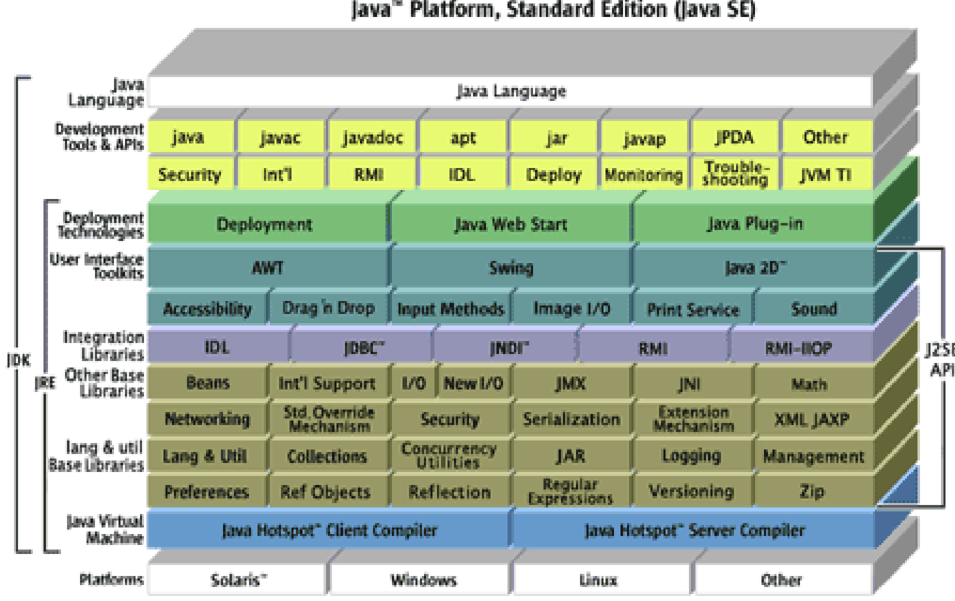


Figura 3.2 - Arquitetura do Java SE e suas tecnologias (SUN, 2006b).

3.2 Java Plataform, Micro Edition (Java ME)

A plataforma Java ME é destinado aos dispositivos com recursos limitados. Esse dispositivos englobam os celulares, PDAs, *set-top boxes* – conversores usados para televisão digital - e outros aparelhos com essas características (SUN MICROSYSTEMS, 2006a). Antes da introdução do Java ME, a maioria destes dispositivos eletrônicos possuía uma natureza estática, por não permitir a instalação de novos *softwares*, além daqueles que foram inseridos no processo de fabricação. Porém, atualmente, tornou-se possível deixá-los muito mais dinâmicos.

Com a introdução do Java em tais dispositivos, tem-se o acesso aos recursos inerentes da linguagem e da plataforma. Isto é, uma linguagem de programação fácil de dominar, um ambiente em tempo de execução que fornece uma plataforma segura e portável, além do acesso ao conteúdo dinamicamente.

Apesar dessas grandes vantagens, a plataforma Java ME ainda possui certas restrições que estão aos poucos sendo sanadas, pois embora fosse desejável ter toda a API Java SE nos

dispositivos micro, isso não é nada realístico hoje. Por exemplo, um celular, com sua tela limitada, não poderia usar todas as APIs de interface gráfica disponíveis na versão do Java para *desktop,* tão quanto suportar as APIs relativas ao desenvolvimento em Java para servidores, que também envolvem questões muito mais complexas.

Além desses fatores, sabe-se que os recursos dentro do Java ME podem variar bastante. Por exemplo, um *pager* possui recursos - tais quais memória, processamento, tamanho físico, de tela, resolução, etc - diferenciados se comparados a um celular. Porém, ainda assim, dentro de um mesmo tipo de dispositivo, pode-se encontrar variações. Por exemplo, apesar de existirem diversos celulares, existem enormes diferenças de recursos entre diferentes modelos de aparelho.

Para resolver esse problema e dar uma maior flexibilidade ao Java ME, foram criados os conceitos de configurações, perfis e pacotes adicionais, conforme ilustrado na Figura 3.3 e descrito nas próximas seções.

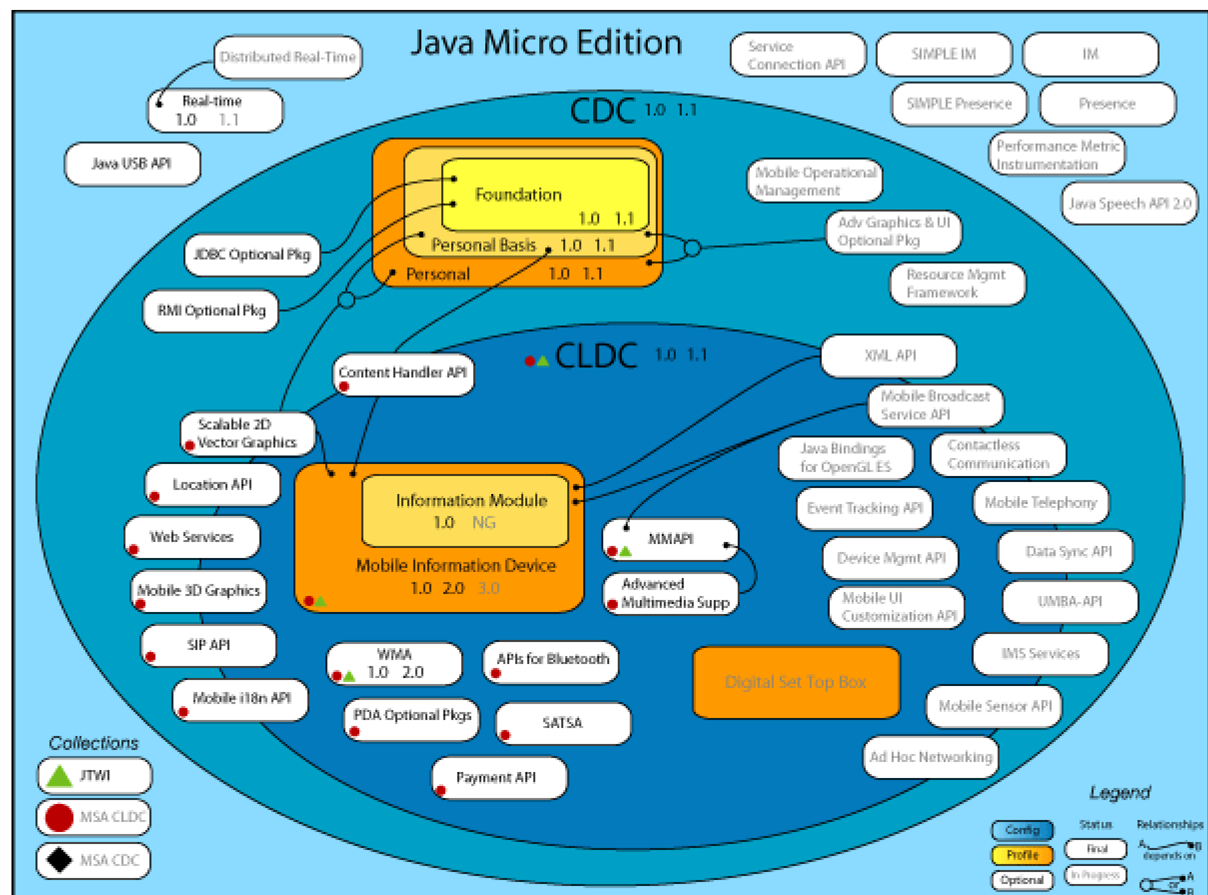


Figura 3.3 - Estrutura da tecnologia Java ME com suas configurações, perfis e pacotes adicionais (SUN, 2006a).

3.2.1 Configurações

Uma configuração define as funcionalidades mínimas para dispositivos com características semelhantes. Sendo assim, mais especificamente, ela define as características quanto à máquina virtual e o conjunto de classes derivadas da plataforma Java SE.

Atualmente, o Java ME divide as configurações em dois tipos, ilustrados na Figura 3.4: configurações mais limitadas, chamadas de CLDC (*Connected Limited Device Configuration*), e configurações com mais recursos, a CDC (*Connected Device Configuration*). A CLDC é um subconjunto da CDC, como mostra a Figura 3.5.

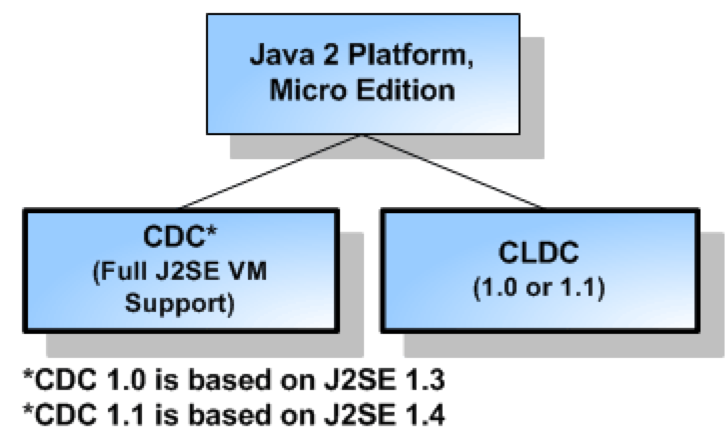


Figura 3.4 – Configurações do Java ME (SUN, 2006e).

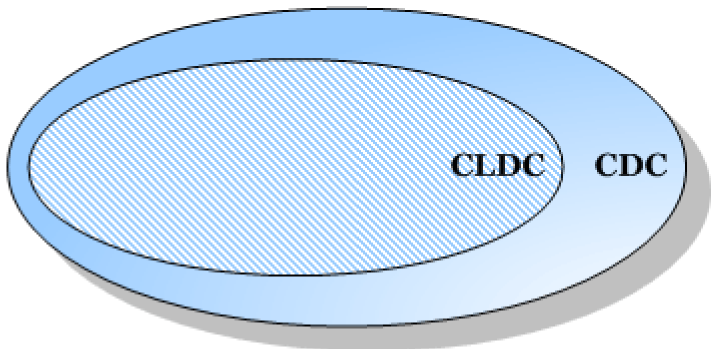


Figura 3.5 – Relacionamento do CDLC perante o CDC (SUN, 2006e).

3.2.2 Generic Connection Framework

Durante o desenvolvimento do Java ME, as APIs java.io e java.net do Java SE foram consideradas um pouco grandes para serem usadas nos dispositivos móveis, então o *Generic Connection Framework* (GCF) foi especificado para suprir essas necessidade.

Como o nome sugere, o *framework* de conexão genérica é usado par fazer diversos tipos de conexões, tais como HTTP, *streams*, datagramas, etc. Ele é baseado na CLDC, entretanto, pelo fato de ser extremamente extensível, atualmente já é possível usá-lo com a configuração CDC através de diversos perfis que o implementam, como também na própria plataforma Java SE, através da especificação JSR 197 (*Generic Connection Framework Optional Package for the J2SE Platform*).

3.2.3 Perfis

Um perfil fornece as bibliotecas para o desenvolvedor poder escrever aplicativos para um determinado tipo de dispositivo. Um perfil poderia ser considerado, analogamente, as características de um dispositivo que pertence a uma família, essa última, representando um tipo de configuração.

Existem diferentes perfis que suprem necessidades distintas nas duas configurações (CDC e CLDC). São eles:

- *Foundation Profile* (FP): É o perfil com menos funcionalidades da configuração CDC. Ele fornece conectividade à rede, mas não fornece classes de interface para construção de aplicações gráficas.
- *Personal Basis Profile* (PBP): Este perfil CDC fornece um ambiente para dispositivos conectados que suportem um nível básico de interface gráfica ou necessitem o uso de toolkits específicos para aplicações.

- *Personal Profile* (PP): Este perfil CDC é utilizado para dispositivos que possuem suporte completo de interface ou applet, como PDAs de alto padrão e consoles para jogos. Ele inclui a biblioteca AWT completa e é fiel ao ambiente web, executando facilmente applets feitos para ambientes desktop.
- *Mobile Information Device Profile* (MIDP): O perfil MIDP é voltado especificamente para dispositivos portáteis, como celulares e PDAs (de baixo desempenho). Este perfil proporciona funcionalidades como: gerenciamento do ciclo de vida de uma aplicação, componentes de interface gráfica e persistência dos dados.
- *Information Module Profile* (IMP): O perfil IMP é direcionado para dispositivos com as características similares àquelas de dispositivos MIDP, mas com quase nenhuma potencialidade de interação com o usuário.

A Figura 3.6 mostra os perfis suportados pela CDC e pela CLDC. Alguns perfis disponíveis para CDC podem ser combinados para aumentar o nível de recursos.

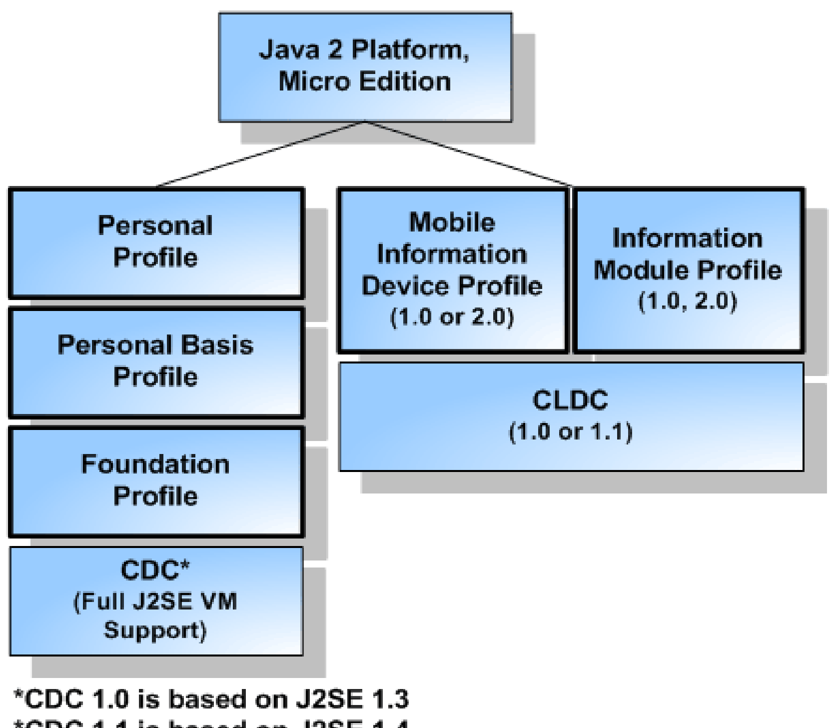


Figura 3.6 - Estrutura dos perfis suportados pela CDC e CLDC (adaptado de SUN, 2006e).

3.2.4 Pacotes Opcionais

A plataforma Java ME pode ser estendida pela combinação de vários pacotes opcionais com as configurações e perfis, conforme ilustrado na Figura 3.7. Criados para atingir requisitos específicos de mercado, os pacotes opcionais oferecem APIs padrões para tecnologias existentes e emergentes, tais como Bluetooth, *web services*, manipulação multimídia, gráficos 3D, envio de mensagens, etc. Eles são opcionais justamente pelo fato de que nem todo dispositivo necessita implementar algum desses pacotes, ou seja, eles são um adicional. Por exemplo, um celular com Bluetooth nativo no qual o fabricante deseje que os desenvolvedores Java utilizem-se dessa tecnologia via *software*, poderia disponibilizar uma implementação do pacote opcional que a define, no caso a JSR 82.

Além dos pacotes opcionais, existem ainda as APIs dos fabricantes dos dispositivos. A diferença entre os dois é que os pacotes opcionais fazem parte de uma especificação da JCP (*Java Community Process*), que é responsável por definir o rumo da tecnologia Java criando especificações padrões para a mesma, enquanto as APIs proprietárias se restringem aos dispositivos daquele fabricante.

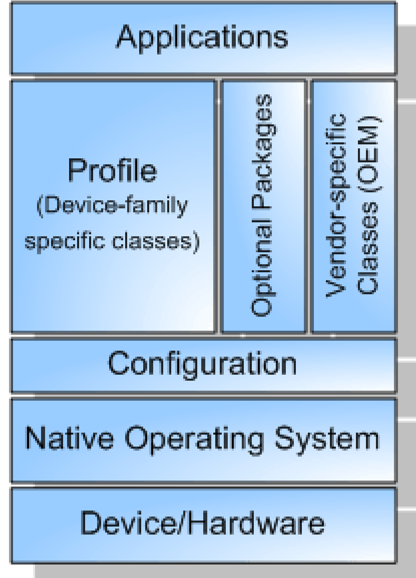


Figura 3.7 - Estrutura mostrando as configurações, perfis, pacotes opcionais e APIs dos fabricantes (SUN, 2006e).

3.3 Java APIs for Bluetooth (JSR 82)

A JSR 82 é um pacote opcional. Ela é o resultado do trabalho de um grupo de especialistas da JCP que definiram essa especificação identificada pelo número 82. Após longos trabalhos, ela foi finalmente concluída ao final de Junho de 2006. Ela utiliza-se do GCF para comunicação Bluetooth. A JSR 82 consiste de dois pacotes listados abaixo.

Pacote	Provê
javax.bluetooth	O core da API Bluetooth
javax.obex	The Object Exchange (OBEX) API

Tabela 3.1 - Pacotes da JSR 82 (adaptado de SUN, 2006e).

O pacote javax.bluetooth provê uma API para busca de dispositivos e pesquisa por serviços. Além disso, também fornce funcionalidades para disponibilizar serviços e a parte da comunicação utilizando o protocolo L2CAP e RFCOMM. Ele é o pacote núcleo dessa especificação.

O pacote javax.obex provê uma API para o protocolo de transferência de objetos (OBEX). Ele é um pacote de implementação opcional, dentro da própria especificação JSR 82, que é um pacote opcional na plataforma Java ME, por isso, atualmente poucos dispositivos implementam ele.

3.4 Implementações da JSR 82

Pelo fato da JSR 82 ter sido criada tendo como base o GCF, ela funciona sobre ele. Logo, é possível criar aplicações que façam uso dela na plataforma Java ME, nas configurações CLDC e CDC (através de diversos perfis) e na Java SE (através da JSR 197).

Existem várias implementações da JSR 82 desenvolvidas em Java SE para funcionarem em diversos sistemas operacionais de computadores pessoais – acessando nativamente a pilha de protocolo Bluetooth disponível. A Tabela 3.2 lista as bibliotecas mais comuns, juntamente com as suas especificações.

Produto	Suporte javax.bluetooth	Suporte javax.obex	Plataformas Java	Sistemas Operacionais	Distribuição
Atinav	Sim	Sim	Java ME, Java SE	Win-32, Linux, Pocket PC	Comercial
Aventana	Sim	Sim	Java SE	Win-32, Mac OS X, Linux, Pocket PC	Freeware para Linux. Comercial
Blue Cove	Sim	Não	Java SE	WinXP SP2	Freeware
Electric Blue	Sim	Sim	Java SE	WinXP SP2	Comercial
Harald	Não	Não	Qualquer plataforma que suporte javax.comm	multi-plataforma	Freeware
JavaBluetooth .org	Sim	Não	Qualquer plataforma que suporte javax.comm	multi-plataforma	Freeware
Rococo	Sim	Sim	Java ME, Java SE	Linux, Palm	Freeware para Linux com fins estudantis. Comercial.

Tabela 3.2 – Implementações da JSR 82 para uso em computadores (adaptado de JAVA BLUETOOTH, 2006).

3.5 Java Virtual Machine

Uma máquina virtual é uma simulação de um *hardware*, contendo suas principais

características, como entrada e saída, interrupções, conjunto de instruções, entre outras.

A máquina virtual Java é a ponte entre a aplicação e a plataforma utilizada, convertendo o *bytecode* da aplicação em código de máquina adequado para o *hardware* e sistema operacional utilizado. Além disso, ela administra a memória do sistema, provendo segurança contra códigos maliciosos, administra as *threads* da aplicação e faz a coleta de objetos sem referências explícitas utilizando-se do G*arbage Collector* (GC). A máquina virtual Java usada na plataforma padrão (Java SE) e na plataforma corporativa (Java EE) foi desenvolvida para ser utilizada em sistemas *desktop* e servidores.

Já na plataforma Java ME deve-se utilizar uma máquina virtual apropriada para os dispositivos com a configuração CLDC e CDC. Em virtude disso, a Sun projetou duas máquinas virtuais:

- *Kilo Virtual Machine* (KVM): É uma nova implementação da JVM, com suporte à configuração CLDC, para dispositivos portáteis com recursos limitados. A KVM foi desenhada para ser pequena, portátil, customizada e para uso de memória estática entre 40Kb a 80Kb. A quantidade mínima de memória requerida pela KVM é 128Kb. O processador pode ser de 16 ou 32 bits.
- *Compact Virtual Machine* (CVM): Foi desenvolvida para adicionar suporte a mais funcionalidades do que a KVM. Com suporte à configuração CDC, esta máquina virtual engloba quase todas as características de uma máquina virtual Java convencional (JVM), só que de forma mais otimizada para dispositivos móveis.

4 FRAMEWORKS

Para FAYAD (1999), um framework é um conjunto de classes que constitui um projeto abstrato para a solução de uma família de problemas. Considerando isso, um *framework* poderia ser mais especificamente definido como uma implementação ou uma estrutura de classes inacabadas com intuito de servir para o desenvolvimento de diferentes aplicações pertencentes a um único domínio.

4.1 Características

Segundo FAYAD (1999), um framework possui algumas características principais como:

- Modularidade – Oferecem modularidade encapsulando detalhes de implementação através de interface, o que facilita a implementação de mudanças distintas.
- Reusabilidade – A estrutura genérica provida pelo *framework* permite reuso para a construção de aplicações diversificadas pertencentes a um mesmo domínio. Como consequência, esse reuso ajudará na produtividade do desenvolvimento, assim como na qualidade, performance, confiança e interoperabilidade do *software*.
- Extensibilidade – Permitem que as aplicações extendam classes e sobrecarreguem métodos facilmente, o que caracteriza a sua facilidade de customização de aplicações desenvolvidas com ele.
- Inversão de Controle (*Inversion of Control* ou comumente chamada apenas de IoC) – É a arquitetura característica de um *framework*. O controle de uma aplicação é a seqüência de chamada de métodos e ações que são executadas. Quando se fala em inversão de controle, significa que a aplicação que use um framework não controlará essas seqüências, delegando a responsabilidade disso ao *framework*.

4.2 Classificação

Segundo FAYAD (1999), *frameworks* podem ser classificados segundo ao seu escopo e tipo de extensão.

Quanto ao escopo, podem ser de três tipos: de infra-estrutura (*System Infraestructure Frameworks)*, de integração de middleware (*Middleware Integration Frameworks)* e de aplicações corporativas (*Enterprise Application Frameworks)*.

Os *frameworks* de infra-estrutura são desenvolvidos com o objetivo de simplificar o desenvolvimento de sistemas de infra-estrutura portáveis e eficientes, tais como sistemas operacionais, *frameworks* de comunicação, de interface gráfica e de processamento de linguagem.

Os *frameworks* de integração de middleware caracterizam-se por serem usados para integrar aplicações distribuídas e componentes. Estes *frameworks* escondem o baixo nível da comunicação entre componentes distribuídos, possibilitando que os desenvolvedores trabalhem em um ambiente distribuído de forma semelhante a que trabalham em um ambiente centralizado. São exemplos de frameworks de integração de middleware Java RMI (Remote Method Invocation) e frameworks ORB (*Object Request Broker*).

Os *frameworks* de aplicações coorporativas são voltados para um domínio de aplicação específico, como por exemplo, os domínios da aviação, telecomunicações e financeiro. Frameworks de aplicações corporativas são mais caros de serem desenvolvidos ou comprados quando comparados com os de integração de middleware e de infra-estrutura, pois são necessários especialistas do domínio de aplicação para construí-los.

Além disso, os *frameworks* também podem ser classificados pelo tipo de extensão, que pode ser basicamente de três formas. A primeira delas seria através da herança entre classes do *framework* e aplicação, conhecida como reuso do tipo caixa branca (*White-Box*). A segunda maneira seria pela configuração e composição entre as classes da aplicação e do *framework*, chamado de caixa preta (*Black-Box*). E a última seria por uma combinação dessas duas últimas formas, que seriam os chamados caixa cinza (*Grey-Box*).

Os *frameworks* caixa branca são dirigidos a arquitetura, logo o desenvolvimento de aplicações em cima deles é feito pela especialização de classes e sobrecarga de métodos. Já os

frameworks caixa preta são dirigidos a dados e o seu reuso é feito através de composição e implementação de interfaces pré-definidas. Os *frameworks* do tipo caixa-cinza são uma combinação das possibilidades anteriores. Apresentam inicialmente partes prontas, permitindo que sejam inseridas novas funcionalidades a partir da criação de classes. Este tipo de *framework* não somente fornece subclasses concretas, mas permite a criação de novas subclasses concretas.

4.3 Desafios relacionados

Segundo FAYAD (1999), existem diversos desafios - como o esforço de desenvolvimento, curva de aprendizado, integração, manutenção, eficiência, validação e remoção de erros e falta de padrões - para adoção de *frameworks* e é preciso medi-los antes de tomar alguma decisão de uso.

O primeiro deles, seria o do esforço de desenvolvimento. Ele ocorre devido à alta complexidade envolvida no projeto de um framework, pois desenvolver *frameworks* que sejam de alta qualidade, extensíveis e reusáveis é uma tarefa muito difícil.

Quanto à curva de aprendizagem, antes de usar um *framework*, é necessário um estudo em cima dele para compreender a sua arquitetura e poder torná-lo útil de fato. Logo, é importante que se gaste menos tempo desenvolvendo a aplicação usando o *framework* do que se não fosse usá-lo. Com o tempo de uso, é provável que essa produtividade aumente, mas é extremamente necessário medir o quão vantajoso e útil ele será.

Outro ponto importante seria o desafio da integração que ocorre quando diversos *frameworks* são usados com outras tecnologias e com sistemas não previstos pela sua arquitetura. Antes de usar um *framework*, o desenvolvedor deve analisar bem as suas características juntamente com os possíveis problemas associados.

Além desses desafios, as aplicações usando *frameworks* podem apresentar um desempenho inferior se comparados a aplicações específicas. Isso ocorre pois, para os *frameworks* tornarem-se extensíveis, são adicionados níveis de indireção representando ainda mais

camadas, e por conseqüência, mais lógica a ser processada. Desta forma, o desenvolvedor de aplicações deve considerar se os ganhos advindos do reuso compensam uma potencial perda de desempenho.

Frameworks inevitavelmente precisam ser mantidos através de correção de erros de programação, acréscimo ou remoção de funcionalidades e de pontos de extensão, além de outras coisas que levam a um gasto de tempo e dinheiro. Em alguns casos, a própria instância do *framework* sofre manutenção para acompanhar a evolução desse processo, e aí cabe ao desenvolvedor da aplicação considerar o esforço de manutenção necessário para obter os benefícios dessa nova versão do *framework*.

Outro ponto interessante, é a validação e remoção de erros de um *framework* ou de uma aplicação que o use, pois pelo fato do *framework* ser uma aplicação inacabada, torna-se difícil - tanto para os mantenedores de *frameworks* quando para os desenvolvedores de aplicação - testar, respectivamente, o *framework* e a instância do *framework* isoladamente. Junto a isso, aplicações que usam *frameworks* costumam ser mais difíceis de depurar, pois devido à inversão de controle, o controle da execução oscila entre a aplicação e o *framework*.

Por fim, ainda não há um padrão amplamente aceito para desenvolver, documentar, implementar e adaptar *frameworks*. E sendo assim, todos os papéis envolvidos no desenvolvimento e uso do framework são afetados por isso, a exemplo do mantenedor do *framework*, que precisa ter um conhecimento profundo sobre os componentes internos e suas interdependência,s e o desenvolvedor de aplicações, que precisa entender o funcionamento do *framework* apesar da falta de padrão das documentações que são geradas.

5 PROJETO MARGE

O projeto Marge[1] é um *framework* desenvolvido em Java - que utiliza a biblioteca JSR 82 – com intuito de ajudar a criação de aplicações que façam uso da tecnologia Bluetooth. A idéia principal do projeto é reduzir a curva de aprendizado para desenvolvedores que pretendem criar aplicações com Bluetooth com objetivo de simplificar essa tarefa, fazendo-os concentrar o foco do desenvolvimento apenas na parte lógica do *software*. O *framework* se responsabilizará por abstrair as outras partes relacionadas à comunicação Bluetooth – como as conexões, protocolos, trocas de mensagens, pesquisa por dispositivos e serviços – de uma maneira fácil e suportando diferentes configurações. O projeto Marge não propões apenas uma biblioteca para simplificar o uso de Bluetooth em Java, mas sim toda uma arquitetura definida pelo *framework*.

Abaixo encontra-se a imagem detalhada de onde se dá o uso do Marge na arquitetura para desenvolvimento de aplicações com Bluetooth. Na primeira camada – de baixo para cima - temos o sistema operacional, seguido de uma pilha de protocolos Bluetooth, e de uma camada de integração entre a máquina virtual Java e a chamada para a pilha. Acima da máquina virtual Java está a JSR 82, a API que especifica o Bluetooth sobre a qual o Marge foi desenvolvido. Acima do Marge, encontram-se as aplicações.

[1] O nome tem origem na personagem do desenho animado do original em inglês 'The Simpsons' © 20th Century Fox Television, criado por Matt Groening,, chamada Marge. Ela é a mãe da família e possui um cabelo azul encaracolado, que é utilizado no projeto para fazer uma analogia a uma rede Bluetooth.

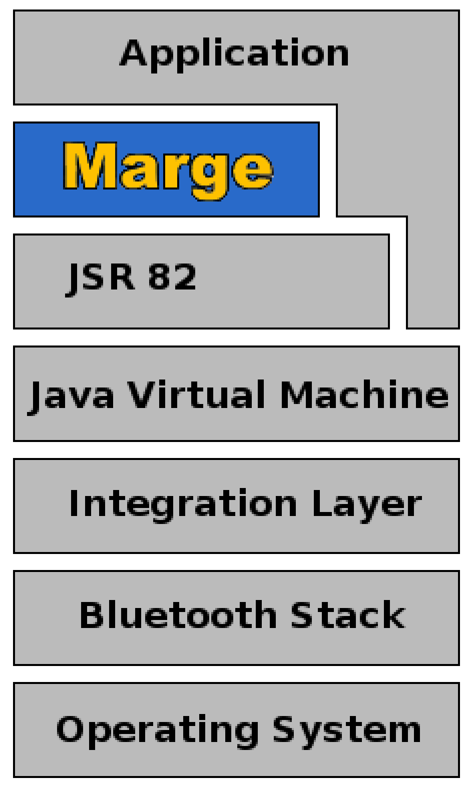


Figura 5.1 – Arquitetura onde o Marge está inserido.

O projeto Marge é livre e possui o código aberto registrado sobre a licença LGPL (*GNU Lesser General Public License)*. Ele está sendo hospedado pelo portal java.net, no endereço http://marge.dev.java.net. O projeto faz parte da comunidade *Mobile and Embedded* desse portal, que é responsável pelos projetos de código livre em Java ME. O logotipo do projeto é mostrado na figura 5.1.

Figura 5.2 – Logo atual do projeto Marge.

5.1 Metodologia de desenvolvimento

Em virtude do curto prazo de desenvolvimento do projeto, serão adotadas algumas práticas de desenvolvimento de metodologias ágeis, mais precisamente do *Extreme Programming* (XP). O XP, assim como outras metodologias ágeis, baseia-se, entre outras coisas, na capacidade de permitir facilmente as mudanças de requisitos, de considerar ciclos iterativos de desenvolvimento menores, possuir a figura do cliente (no caso o orientador do trabalho), junto como membro da equipe de desenvolvimento e buscar refatoramentos de código constantes para melhoria da estrutura interna do *software* em questão.

O XP será bastante adequado para o desenvolvimento do projeto, pois se pretende incrementalmente ir adicionando funcionalidades ao *framework*, à medida que são desenvolvidas aplicações de testes e efetuado os refatoramentos.

5.2 Ferramentas de desenvolvimento

Será utilizada a IDE (*Integrated Development Envionment*) Eclipse 3.2 juntamente com o seu plugin para o desenvolvimento Java ME chamado de EclipseME 1.5.5 e os kit de desenvolvimento para dispositivos móveis da Sun Microsystems, J2ME WTK (*Wireless Toolkit*) 2.2, que emula dispositivos móveis genéricos. Além disso, será também utilizado o sistema de controle de versão *Subversion*, popularmente chamado de SVN, disponível na rede INF da UFSC. Ao final do projeto, o repositório será migrado para o SVN do portal java.net.

Para documentação UML, será utilizada a ferramenta ArgoUML, versão 0.24, que é desenvolvida em Java e possui o código aberto. Além disso, também será feita a engenharia reversa das classes no plugin UML do Netbeans 5.5 para a geração de alguns diagramas de classes automaticamente.

5.3 Classificação do framework

O *framework* Marge pode ser classificado como do tipo de infra-estrutura quanto ao escopo, por ser idealizado para simplificar o processo de conexão, comunicação e suportar diferentes configurações para o desenvolvimento de aplicações desse determinado contexto.
Quanto a extensão, o mesmo pode ser classificado como do tipo caixa cinza, pois oferece uma arquitetura pronta com subclasses concretas. Entretanto, ele também permite a extensão para fins variados, como, por exemplo, para a criação de um novo protocolo de comunicação específico.

5.4 Estrutura do framework

O código foi desenvolvido em inglês para facilitar a integração futura de desenvolvedores da comunidade de software livre.
Atualmente, o *framework* está na versão 0.4.0, que é a caracterizada por ser a segunda versão lançada, a anterior era a 0.3.4. Essa última versão apresenta um tamanho de 24730 *bytes* e sua estrutura geral será detalhada neste tópico.

5.4.1 Pacotes e classes

A versão 0.4.0 do *framework* Marge contém 20 classes divididas em 5 pacotes. Cada pacote

irá ser detalhado abaixo juntamente com o seu diagrama de classes.

O pacote *net.java.dev.marge.communication* contém a estrutura básica para comunicação. Este pacote contém as seguintes classes, conforme ilustrado na Figura 5.3:

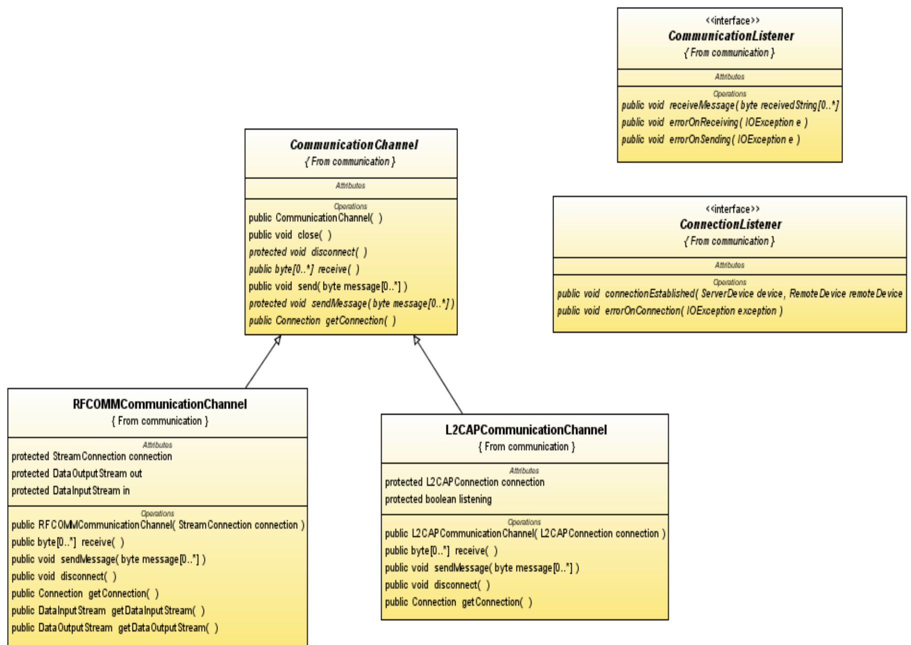


Figura 5.3 – Diagrama de classes do pacote *net.java.dev.marge.communication*.

- *CommunicationChannel*: Interface que representa um canal de comunicação e define métodos relativos a manipulaçao da conexão, recebimento e envio de mensagens.
- *CommunicationListener*: Interface *listener*[2] que define métodos para serem notificados relativos ao processo de comunicação: fechamento da conexão, recebimento de mensagens, erros durante o recebimento e troca de mensagens.
- *ConnectionListener*: Interface *listener* que define métodos para serem notificados relativo ao processo de conexão: quando a mesma é estabelecida ou quando acontece um erro durante ela.
- *L2CAPCommunicationChannel*: Classe que representa um canal de comunicação usando o protocolo L2CAP.
- *RFCOMMCommunicationChannel*: Classe que representa um canal de comunicação usando o protocolo RFCOMM.

---

[2] Listener é o nome usual do padrão de projeto Observer ,criado pelo GoF (*Gang of Four*), na qual se baseia no uso de produtores de eventos que chamam os determinados consumidores registrados desses respectivos eventos.

O pacote *net.java.dev.marge.entity* contém as entidades básicas do *framework*, mostradas na Figura 5.6. Tais entidades são:

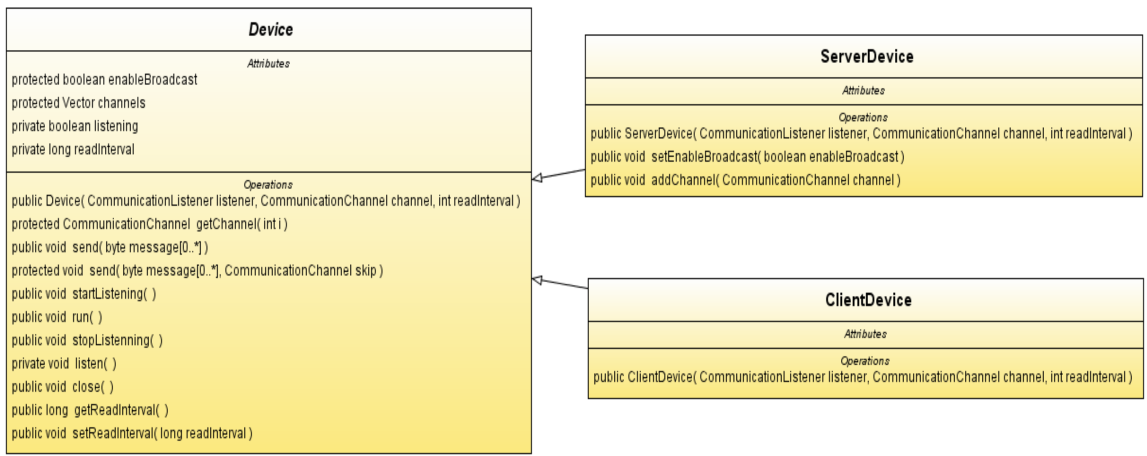


Figura 5.4 – Diagrama de classes do pacote *net.java.dev.marge.entity*.

- *ClientDevice*: Classe que representa um dispositivo cliente.
- *Device*: Classe abstrata que define a estrutura de um dispositivo genérico.
- *ServerDevice*: Classe que represent um dispositivo servidor.

O pacote *net.java.dev.marge.entity.config* contém as configurações necessárias para que o *framework* instancie um cliente ou servidor. No cliente, entre outras configurações, é possível definir qual será o intervalo para que ele leia as mensagens que forem eviadas à ele e qual será o serviço na qual ele irá se conectar. No servidor, também é possível especificar o intervalo de leitura, como também o serviço que ele disponibilizará, o número máximo de conexões aceitas, o nome dele e alguns parâmetros de segurança, como encriptação, autenticação e autorização. O *framework* não possui controle sobre funcionamento dessa segurança, pois como ele trabalha acima da JSR 82, quem deve garantir isso será o dispositivo que o implementar. A Figura 5.7 mostra as classes desse pacote, que são descritas a seguir:

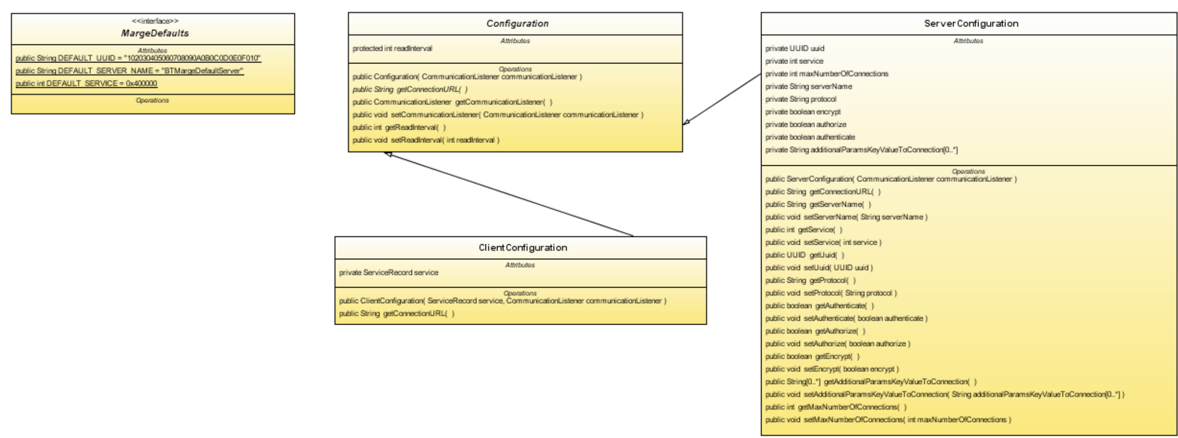


Figura 5.5 – Diagrama de classes do pacote *net.java.dev.marge.entity.config*.

- *ClientConfiguration*: Classe que representa a configuração de um cliente.
- *Configuration*: Classe abstrata de configuração genérica.
- *MargeDefaults*: Interface que define algumas constantes com valores padrão para configuração do servidor e cliente.
- *ServerConfiguration*: Classe que representa a configuração de um servidor.

O pacote *net.java.dev.marge.factory* contém as fábricas[3], responsáveis pela instanciação de clientes e servidores para cada protocolo. O conteúdo do pacote é ilustrado pela Figura 5.8, e suas classes são assim descritas:

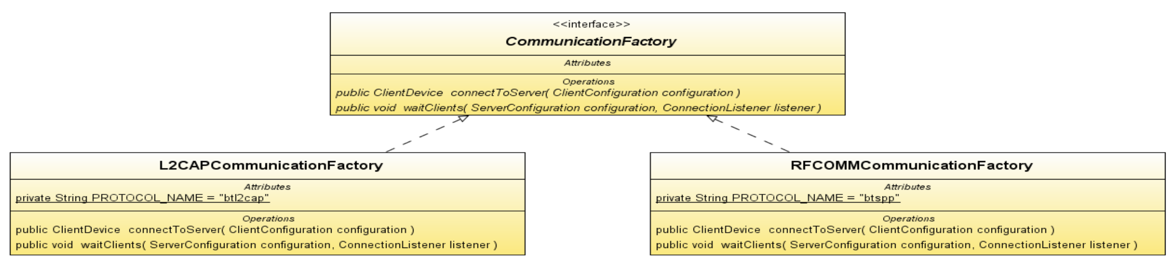


Figura 5.6 – Diagrama de classes do pacote *net.java.dev.marge.factory*.

[3] Padrão de projeto na qual a classe fábrica é responsável por encapsular a criação de instâncias de outra classe.

- *CommunicationFactory*: Interface que define métodos para criação de dispositivos clientes e servidores.
- *L2CAPCommuni*cationFactory: Classe que define métodos para criação de dispositivos clientes e servidores que usam o protocolo L2CAP para comunicação.
- *RFCOMMCommunicationFactory*: Classe que define métodos para criação de dispositivos clientes e servidores que usam o protocolo RFCOMM para comunicação.

O pacote *net.java.dev.marge.inquiry* contém as classes especializadas pelas buscas de dispositivos e serviços que envolvem o protocolo SDP, utilizado para descoberta de serviços. A Figura 5.9 apresenta o diagrama de classes deste pacote. Suas classes são as seguintes:

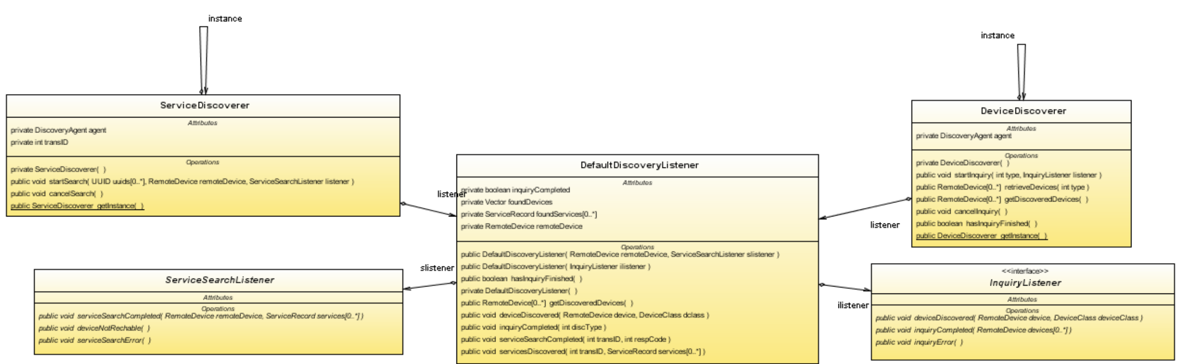


Figura 5.7 – Diagrama de classes do pacote *net.java.dev.marge.inquiry*.

- *DefaultDiscoveryListener:* Classe *listener* que contém a implementação padrão do *framework* para a busca por dispositivos e pesquisa por serviços.
- *DeviceDiscoverer*: Classe *singleton*[4]. que fornece métodos para a busca por dispositivos.
- *InquiryListener*: Interface *listener* que contém métodos relativos ao processo de busca por dispositivos.
- *ServiceDiscoverer*: Classe *singleton* que fornece métodos para a pesquisa por serviços.
- *ServiceSearchListener*: Interface *listener* que contém métodos relativos ao processo de pesquisa por serviços.

5.4.2 Diagrama de classes

4 Padrão de projeto usado quando se deseja que uma determinada classe tenha apenas uma instância.

O diagrama de classes do Marge 0.4.0, contendo todos os pacotes comentados anteriormente, está representado abaixo.

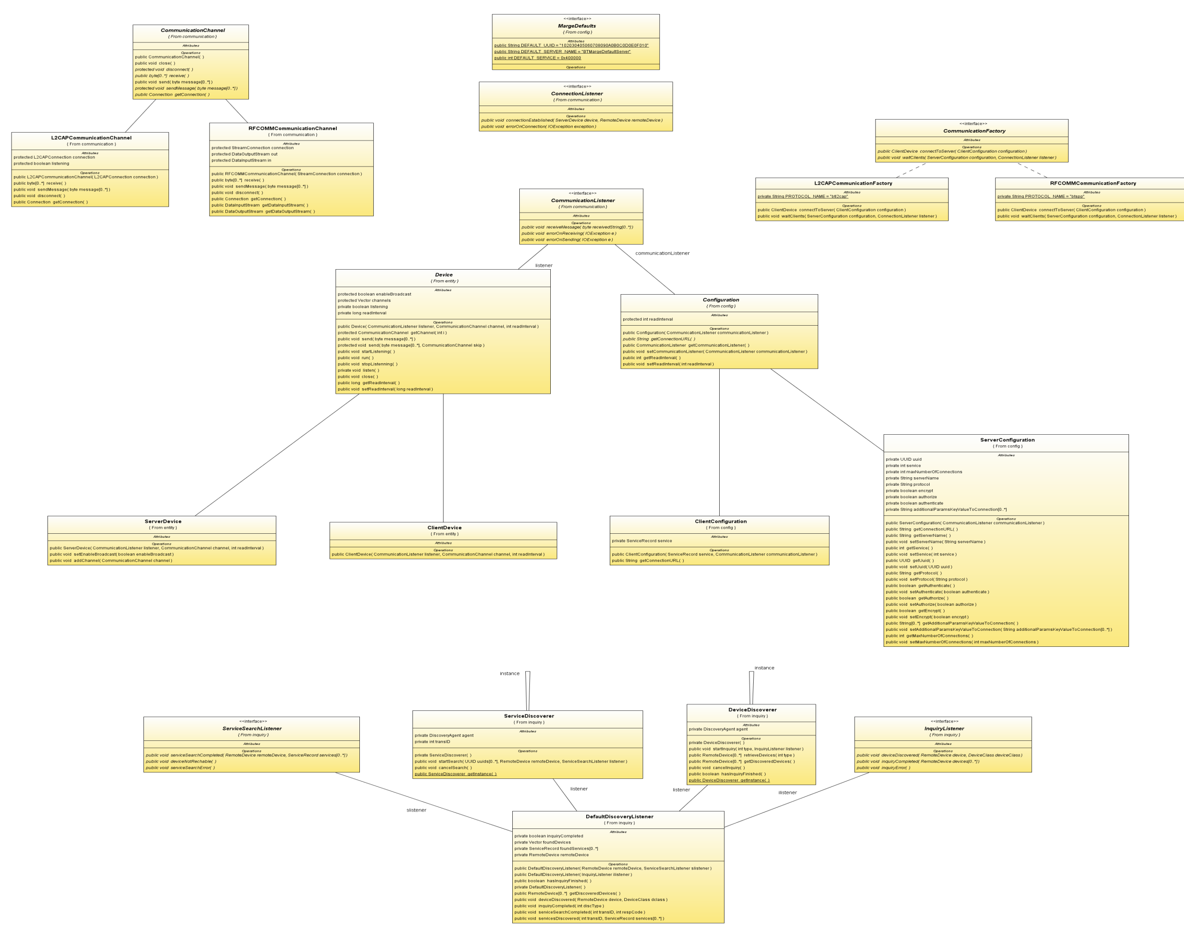


Figura 5.8 – Diagrama de classes da versão 0.4.0 do Marge.

5.5 Documentação

Foi gerada uma documentação inicial, com guias de como construir uma aplicação utilizando o Marge e a mesma foi disponibilizada no *site* e no *wiki*[5].

[5] Software colaborativo que permite a edição coletiva dos documentos usando um sistema que não necessita que o conteúdo tenha que ser revisto antes da sua publicação.

(http://wiki.java.net/bin/view/Mobileandembedded/Marge) do projeto. Documentação é fundamental em projetos de *software* livre, pois os desenvolvedores necessitam aprender como utilizar a ferramenta e também ter um suporte à ela. O portal java.net também oferece outras ferramentas interessantes para facilitar esse processo, como fóruns e listas de email, que também estão sendo utilizadas.

5.5.1 Javadoc

O Javadoc da versão 0.4.0 do Marge está disponível no *wiki* do projeto.

5.5.2 Forma de utilização do Framework

O processo de estabelecer uma conexão Bluetooth entre dispositivos é feito utilizando o modelo cliente/servidor. Para criar uma instância de um servidor - representado pela classe *ServerDevice* do pacote *net.java.dev.marge.entity* - ou cliente - representado pela classe *ClientDevice* desse mesmo pacote - primeiramente é necessário definir o protocolo de comunicação a ser adotado. Caso se decida pelo RFCOMM, todas as instâncias criadas, do tipo cliente ou servidor, serão instanciadas através da fábrica *RFCOMMComunicationFactory*; caso seja a L2CAP, será usada a fábrica *L2CAPComunicationFactory*. As fábricas pertencem ao pacote *net.java.dev.marge.factory* e possuem os métodos *connectToServer,* que retorna um *ClientDevice,* e *waitClients*, que retornará o *ServerDevice* e o *ClientDevice,* que se conectou, de uma outra forma que será detalhada abaixo.

Cada um desses métodos recebe como parâmetro uma instância concreta da classe *Configuration* do pacote *net.java.dev.marge.entity.config*, podendo ser uma *ServerConfiguration* ou *ClientConfiguration.* Inicialmente, essas configurações já possuem alguns valores padrões para funcionar, porém eles podem ser alterados manualmente, caso necessário. No caso do método *waitClients*, ele também irá receber uma instância que implementa a interface *ConnectionListener,* que será notificada quando alguma conexão for

estabelecida, retornando, então, os dispositivos envolvidos, tanto o servidor, como o cliente.
Um *ServerConfiguration* deve obrigatoriamente receber, no construtor, um *CommunicationListener*, que é uma interface pertencente ao pacote *net.java.dev.marge.communication* que define métodos para tratamento das seguintes situações: quando uma mensagem é recebida; quando ocorre um erro na recepção de uma mensagem; quando ocorre um erro no envio de uma mensagem.
Um *ClientConfiguration* deve também obrigatoriamente receber um *CommunicationListener* no construtor, mas também uma intância da classe *ServiceRecord,* que é uma classe da própria JSR 82, do pacote *javax.bluetooth*, e representa um serviço, no caso, o serviço na qual o cliente deseja se conectar.
Para o cliente realizar a busca por dispositivos e a pesquisa por serviços em um dispositivo encontrado e assim, obter uma intância do *ServiceRecord* desejado, é necessário o uso do pacote *dev.marge.net.inquiry*. Esse pacote contém a classe *DeviceDiscoverer* que realiza a busca por dispositivos e a classe *ServiceDiscoverer* que realiza a procura por serviços em um determinado dispositivo encontrado.

Abaixo seguem dois diagramas de sequência que descrevem o estabelecimento com sucesso de uma conexão entre servidor e cliente, utilizando o protocolo de comunicação RFCOMM:

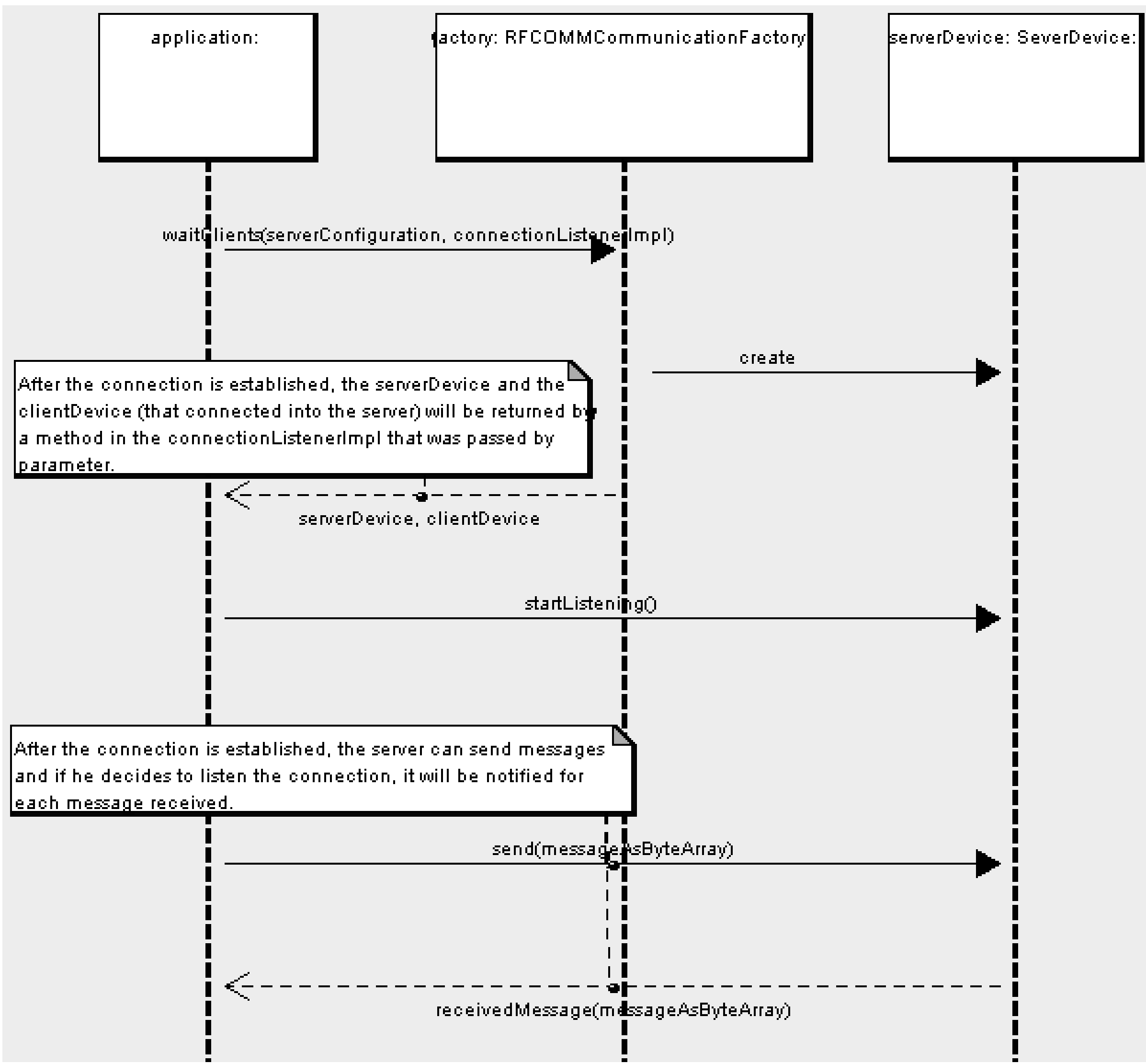


Figura 5.9 – Diagrama de seqüência simplificado do caso mais comum do processo de uso de um servidor RFCOMM.

Um servidor RFCOMM é criado através da configuração mandada por parâmetro para a fábrica. Quando alguém se conectar à ele, a instância *connectionListenerImp*, que implementa a interface *net.java.dev.marge.communication.ConnectionListener*, irá retornar esse servidor, juntamente com a instância do dispositivo cliente que se conectou. Após a conexão estar estabelecida, o servidor inicia a escuta para poder ser avisado pela interface *CommunicationListener* – que foi previamente na configuração do servidor, pela instância *serverConfiguration* - do recebimento de mensagens e também poderá enviar normalmente mensagens. Em seguida são mostrados alguns trechos de código que demonstram como o *framework* é utilizado pelo servidor.

```
// Criando uma fábrica utilizando o protocolo RFCOMM
CommunicationFactory factory = new RFCOMMCommunicationFactory();

// Criando a configuração do servidor
ServerConfiguration sconf = new ServerConfiguration(new
CommunicationListenerImpl());

// Instanciando o servidor via fábrica
factory.waitClients(sconf, new ConnectionListenerImpl());

// Iniciando a escuta no servidor
serverDevice.startListening();

// Servidor enviando uma mensagem
serverDevice.send("Test message".getBytes());

// Fechando a conexão
serverDevice.close();
```

Figura 5.10 – Exemplo de código do Servidor.

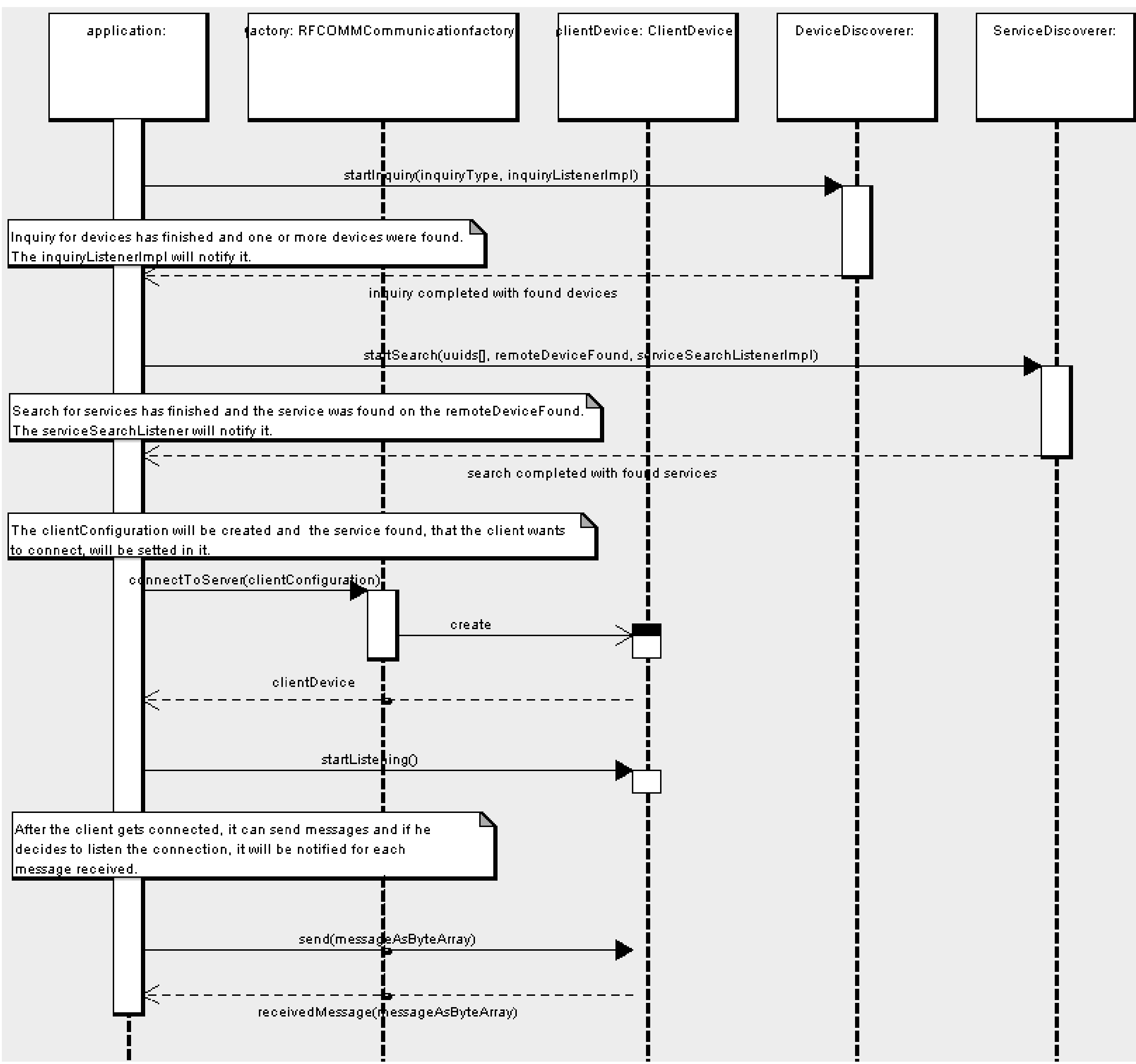


Figura 5.11 – Diagrama de sequência simplificado do caso mais comum do processo de uso de um cliente RFCOMM.

Inicialmente, se faz uma busca por dispositivos com Bluetooth ligado utilizando a classe *net.java.dev.marge.inquiry.DeviceDicoverer*. O *inquiryListenerImpl*, definido pela interface *net.java.dev.marge.inquiry.InquiryListener*, retorna os dispositivos encontrados. Depois faz-se uma busca pelos serviços desejados no dispositivo encontrado escolhido utilizando a classe *net.java.dev.marge.inquiry.DeviceDicoverer*. O serviceSearchL*istenerImpl*, definido pela interface *net.java.dev.marge.inquiry.ServicesSearchListener*, retorna o serviço, caso ele seja encontrado. Com esse serviço, o dispositivo cliente pode ser criado através da fábrica RFCOMM passando como parâmetro uma configuração, que contém, entre outras coisas, o

serviço a qual ele deseja se conectar. Após a conexão, o cliente pode enviar normalmente mensagens para o servidor e caso ele inicie a escuta, ele será também notificado, por um *listener,* que foi definido na configuração*,* das mensagens que forem recebidas. A seguir são apresentados alguns trechos de código relacionados ao uso do *framework* no cliente.

```
// Iniciando a busca por dispositivos
DeviceDiscoverer.getInstance().startInquiry(DiscoveryAgent.GIAC, new
InquiryListenerImpl());
// Iniciando a busca por serviços em um determinado dispositivo encontrado
ServiceDiscoverer.getInstance().startSearch(uuidArray, remoteDeviceFound, new
ServiceSearchListenerImpl());
// Criando uma fábrica utilizando o protocolo RFCOMM
CommunicationFactory factory = new RFCOMMCommunicationFactory();
// Criando a configuração do cliente
ClientConfiguration cconfig = new ClientConfiguration(serviceRecord, new
CommunicationListenerImpl());
// Instanciando o cliente via fábrica
ClientDevice clientDevice = factory.connectToServer(cconf);
// Iniciando a escuta no cliente
clientDevice.startListening();
// Cliente enviando uma mensagem
clientDevice.send("Test message".getBytes());
// Fechando a conexão
clientDevice.close();
```

Figura 5.12 – Exemplo de código do Cliente.

5.6 Aplicações desenvolvidas

Foram desenvolvidas quatro aplicações exemplos para validar o uso do *framework* e ajudar no seu refatoramento durante o processo de desenvolvimento. O código das aplicações está disponível no *site* do projeto.

5.6.1 Chat (*Blue Chat*)

Foi desenvolvida uma aplicação de chat simples em Java ME que comporta até 8 dispositivos. A aplicação baseava na escrita de mensagens e envio outros dispositivo. Ela funciona sobre os protocolos RFCOMM ou L2CAP, permitindo que o usuário escolha o protocolo ao iniciar a aplicação. Foi definido um protocolo simples baseado em Strings para a troca de informação.

Ela é composta de 7 classes, detalhadas abaixo, e dessas, só a *ChatRoom*, *InquiryScreen* e *MainMenu* fazem uso do *framework*.

- *ChatMIDlet*: Classe principal.
- *AboutScreen*: Tela de informações que mostra um texto sobre o Projeto Marge e sobre a aplicação Blue Chat.
- *ChatRoom*: Tela principal de uso em que se faz possível enviar as mensagens e recebê-las.
- *InquiryScreen*: Tela de busca de dispositivos.
- *LoadingScreen*: Tela de espera de conexões quando se cria um servidor.
- *MainMenu*: Menu principal com as opções para criar um servidor ou cliente.
- *ImageUtil*: Classe utilitária usada para carregar as imagens mostradas no AboutScreen.

A imagem de sua interface, encontra-se na Figura 5.13 e o da aplicação encontra-se em anexo.

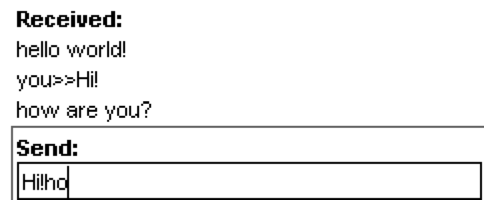


Figura 5.13 – Interface gráfica do chat.

5.6.2 Jogo da Velha (*Tik Tak Tooth*)

Foi desenvolvido um jogo da velha, ilustrado na Figura 5.14, que funciona entre dois dispositivos Java ME. O jogo possui algumas imagens padrões e funciona nos moldes de um jogo da velha tradicional. O processamento lógico é feito em cada dispositivo e o jogo baseia-se em turnos. Foi definido um protocolo simples baseado em Strings para a troca de informação.

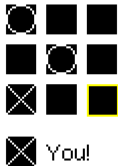


Figura 5.14 – Interface gráfica do Jogo da Velha.

5.6.2 Pong (*Blue Pong*)

Foi desenvolvido um jogo em Java ME no estilo do clássico Pong, chamado Blue Pong, que funciona entre dois jogadores. O objetivo do jogo é acertar a esfera com sua barra horizontal e lançá-la à tela adversária. O processamento do jogo é feito no dispositivo servidor, e enviado o retorno deste ao outro para atualização, pois o mesmo funciona em tempo real. Foi definido um protocolo simples baseado em Strings para troca de informação.
Esse jogo foi desenvolvido com a ajuda do aluno de ciências da computação da UFSC, Lucas Torri, que será um dos futuros mantenedores do projeto Marge em conjunto com seu autor.

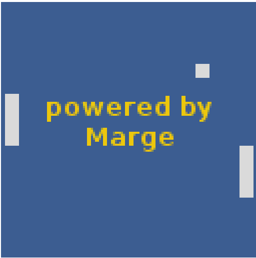


Figura 5.15 – Interface gráfica do Pong.

5.7.4 Passador de slides para o Impress da suite Open Office

Foi desenvolvido um passador de *slides* para o Impress da suite Open Office. Foi utilizada a API UNO (*Universal Network Objects*) para a comunicação com o Open Office. O software divide-se em dois módulos: um servidor Java SE e um cliente Java ME. Foi definido um

formato de arquivo XML (*Extensible Markup Language*) para integração dos aplicativos utilizando o protocolo de comunicação RFCOMM. Para a parte do cliente, também foi necessário a utilização de uma biblioteca de código aberto para o processamento do XML, chamada de kXML.

O passador de slides será reescrito para tornar-se uma extensão integrada da suite e migrado para o projeto mOOo (mobile OpenOffice.org), hospedado no java.net, no endereço htp://mooo.dev.java.net.

5.8 Testes

Foram realizados testes de funcionamento das aplicações, citadas anteriormente, em emuladores e dispositivos celulares, para as aplicações Java ME, e em computadores com a arquitetura IBM PC, para aplicações Java SE. Os testes baseavam-se no uso da aplicação e averiguação de seu funcionamento correto.

Não foram realizados testes de performance sobre o uso do *framework*.

5.8.1 Emuladores

Para testes com as aplicações Java ME foi utilizado o J2ME WTK 2.2 da Sun Microsystems que emula também conexões Bluetooth.

Figura 5.16 – J2ME Wireless Toolkit 2.2.

5.8.2 Celulares

Os testes nos dispositivos celulares foram feitos utilizando os modelos da Nokia 6230, 6111 e 6681, além dos Sony Ericsson W300 e W800. Os celulares não apresentaram problemas e funcionaram conforme o emulador. Abaixo seguem as imagens dos dispositivos usados.

Figura 5.17 – Nokia 6230.

Figura 5.18 – Nokia 6111.

Figura 5.19 – Nokia 6681.

Figura 5.20 – Sony Ericsson W300.

Figura 5.21 – Sony Ericsson W800.

5.8.3 Computadores

Foram feito testes em computadores com o sistema operacional Windows, utilizando a pilha de protocolos Bluetooth do sistema e a biblioteca Bluecove (BLUECOVE, 2006), versão 1.2.2, que implementa parcialmente a JSR 82 (não implementa o pacote *javax.obex* e alguns métodos do *javax.bluetooth*), porém possui o código aberto.

Além disso, também foram feito testes no sistema operacional Linux, utilizando a pilha de protocolos Bluetooth chamada BlueZ (BLUEZ, 2006), que está disponível no kernel, e a biblioteca Avetana (AVENTANA, 2006), versão 20060413, que também possui uma implementação de código aberto. Em ambos os testes, foi utilizado o *dongle* USB Bluetooth Classe 1, do fabricante Billionton.

6 CONCLUSÃO

6.1 Considerações Finais

Definitivamente a especificação JSR 82 é caracterizada por possuir um difícil aprendizado. O Projeto Marge surge com intuito de facilitar o desenvolvimento, em Java, de aplicações comuns que usem Bluetooth. Como foi abordado no capítulo 2, o Bluetooth é uma tecnologia sem fio muito usada para conectar dispositivos a curta distância. Além disso, ela passou por um grande amadurecimento ao longo de sua criação e hoje conta com o apoio de grandes empresas. Por outro lado, também é notável o crescimento da linguagem de programação Java em diversas áreas distintas, inclusive em dispositivos móveis e embarcados, onde esta está muito a frente de outras. Sendo assim, nada mais importante que trazer essa facilidade de uso da tecnologia Bluetooth aos desenvolvedores da plataforma Java, através do projeto Marge. Com isso, poder-se-á explorar esse mercado que ainda está em ascensão e que tanto necessita da geração de produtos e serviços que usem novos paradigmas de comunicação sem fio.

O projeto Marge é relativamente novo, mas está tendo uma boa repercussão na área (ver anexo A). Como todo projeto *software* livre, ele necessitará da ajuda de diversas pessoas para evoluir, não apenas pessoas que produzam código, mas que também ajudem em outros aspectos que são importantes para um sucesso de um projeto, tais como: documentação, testes, promoção, tradução, entre outros. Após a publicação da versão 0.3.4, o projeto ganhou um novo incentivo, com a ajuda do aluno da UFSC, de ciências da computação, Lucas Torri. Outras pessoas também já se inscreveram no projeto para colaborar, totalizando, atualmente, um grupo com mais de 10 indivíduos. O autor do projeto irá coordenar o desenvolvimento do mesmo. Estão sendo geradas muitas idéias e propostas de melhorias para as próximas versões.

6.2 Trabalhos Futuros

Pode-se dizer que o projeto Marge conclui uma etapa nessa sua fase de criação, mas por se tratar de um *software* livre, ele está sujeito a constantes melhorias em diversos aspectos. O projeto continuará a evolução após a conclusão desse trabalho baseado em alguns pontos que precisam ser melhorados, como por exemplo:

- Aumentar o poder de abstração do *framework* para tornar o desenvolvimento de aplicações ainda mais simples.
- Melhorar a parte de busca por dispositivos e pesquisa por serviços para facilitar a listagem direta de dispositivos com um determinado serviço rodando.
- Implementar o protocolo OBEX no *framework* e a criar uma aplicação exemplo utilizando isso.
- Criar uma extensão, baseada no Marge, para utilização em jogos Java ME.
- Criar extensões, baseadas no Marge, para Java ME e Java SE, podendo, assim, usar todo o poder das duas plataformas.
- Melhorar a documentação.
- Criar testes automatizados.
- Realizar testes de performance no *framework*.
- Realizar testes utilizando outros dispositivos, bibliotecas que implementem a JSR 82 e pilhas de protocolos Bluetooth.
- Internacionalizar/localizar as aplicações exemplos e a documentação.
- Realizar testes integrando aplicações Bluetooth feitas em outras linguagens de programação como, por exemplo, Python.
- Refatorar a aplicação exemplo ‘passador de slides’, para torná-la uma extensão real para o Impress da suite Open Office.

7 REFERÊNCIAS BIBLIOGRÁFICAS

APPLE. **Apple - Bluetooth**. Apple Computer, Inc, 2006a. Disponível em <http://www.apple.com/bluetooth/>. Acesso em 22 de outubro de 2006.

APPLE. **Apple - Bluetooth Device Access Guide**. Apple Computer, Inc, 2006b. Disponível em <http://developer.apple.com/documentation/DeviceDrivers/Conceptual/Bluetooth >. Acesso em 22 de outubro de 2006.

ALESSO, H. Peter e SIMTH, Craig. **The Intelligent Wireless Web**. Addison Wesley Longman, Inc, 2001. 384p.

ALHAKIM, Mohammed Maher; AL-KITTANI, Ibrahim; BAKLEH, Anas; SWIDAN, Mohammed; ZARKA, Nizar Dr. **Bluetooth Remote Control**. Information and Communication Technologies, 2006. Disponível em <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=35471&arnumber=1684832 &count=302&index=129>

AVELINK. **Avelink**. Disponível em <http://www.avelink.com/Bluetooth/>. Acesso em 22 de outubro de 2006.

AVENTANA. **Aventana Bluetooth JSR82 Implementation for Windows, MacOS X and Linux Systems**. Disponível em <http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.eng.xml>. Acesso em 22 de outubro de 2006.

BAKKER, DEE M.; GILSTER, Diane McMichael; GILSTER, Ron. **Bluetooth End to End.** Hungry Minds, 2002. 384p.

BARAEV, Ilya; MITTAL, Nitin. **Mobile Bluetooth Networking With JSR 82:** Practical Recommendations and Advanced Practices. JavaOne. TS-3234. 2006. Disponível em <http://developers.sun.com/learning/javaoneonline/2005/mobility/TS-3234.pdf>. Acesso em 22 de outubro de 2006.

BENHUI. Disponível em <http://www.benhui.net/>. Acesso em 22 de outubro de 2006.

BISDIKIAN, Chatschik e MILLER, Brent A. **Bluetooth Revealed**. Prentice Hall,

2000.

BLIPSYSTEMS. Case Stories. **Blipsystems** Disponível em <http://www.blipsystems.com/Default.aspx?ID=735>. Acesso em 22 de outubro de 2006.

BLUECASTING. **BlueCasting**. Disponível em <http://www.bluecasting.com/>. Acesso em 22 de outubro de 2006.

BLUECOVE. **Bluecove**. Disponível em <http://sourceforge.net/projects/bluecove>. Acesso em 22 de outubro de 2006.

BLUETOOTH MEMBERSHIP. **Bluetooth – The Official Bluetooth Membership Site**. Disponível em <https://www.bluetooth.org/>. Acesso em 22 de outubro de 2006.

BLUETOOTH SIG. **Bluetooth SIG**. Disponível em <https://www.bluetooth.com/>. Acesso em 22 de outubro de 2006.

BLUETOOTH QUALIFICATION. **Bluetooth Qualification Program Website**. Disponível em <http://qualweb.bluetooth.org/Template2.cfm?LinkQualified=QualifiedProducts>. Acesso em 22 de outubro de 2006.

BRAY, Jennifer; SENESE, Brian; MCNUTT, Gordon; MUNDAY, Bill; KRAMMER, David. **Bluetooth Application Developer's Guide.** Syngress Publishing, 2001. 556p

BLUEZ. **BlueZ - Official Linux Bluetooth protocol stack**. Disponível em <http://www.bluez.org>. Acesso em 22 de outubro de 2006.

DREAMTECH SOFTWARE TEAM. **WAP, Bluetooth and 3G Programming:** Cracking the Code. Wiley, John & Sons, Inc, 2001. 552p.

CNET NEWS. **Emerging market fuel cell phone growth | CNET NEWS**. Disponível em <http://news.com.com/2100-1039_3-6159491.htm>. Acesso em 22 de outubro de 2007.

EBEAM PROJECTION. **Luidia**. Disponível em <http://www.e-beam.com/products/projection.html>. Acesso em 22 de outubro de 2006.

ESTADAO. Brasil Teleom começa as vendas do Telefone Único. Estadão. **http://www.estadao.com.br/tecnologia/telecom/noticias/2006/jul/31/92.htm**. Disponível em <

http://www.estadao.com.br/tecnologia/telecom/noticias/2006/jul/31/92.htm>. Acesso em 22 de outubro de 2006.

EWALT, David M. FedEx Bites Into Bluetooth. **Forbes**, 29 out. 2004. Disponível em <http://www.forbes.com/2004/10/29/cx_de_1029fedex.html?partner=telecom_newsletter >. Acesso em 22 de outubro de 2006.

FAYAD, M.E; SCHMIDT, D.C.; JOHNSON, R.E., **Domain-Specific Application Frameworks**. Wiley, 1999.

FRODIGH, M., Johansson, P. and Larsson, P. **Wireless ad hoc networking - The art of networking without a network**. Ericsson Review No. 4, 2002.

FORUM NOKIA. **Forum Nokia**. Disponível em <http://forum.nokia.com>. Acesso em 22 de outubro de 2006.

GANGULI, Madhushree. **Getting Started with Bluetooth**. Thomson Course Technology, 2002. 389p.

GOELZER, Maiquel. Bluecove: Comunicando aplicativos J2SE com aplicativos J2ME através de Bluetooth. **Revista Web Mobile**, Grajaú, v. 2, n. 8, p. 20-27, abr./mai. 2006.

GRATTON, Dean A. **Bluetooth Profiles**. Pearson Education, 2002. 550p.

HAARSEN, Jaap. **BLUETOOTH - The universal radio interface for ad hoc, wireless connectivity**. 1998.

HARALD. **Harald – A Java Bluetooth Stack**. Disponível em <http://www.control.lth.se/%7Ejohane/harald/>. Acesso em 22 de outubro de 2006.

HARTE, Lawrence. **Introduction to Bluetooth:** Technology, Market, Operation, Profiles, and Services. Althos, 2004. 64p.

HELD, Gilbert. **Data over Wireless Networks: Bluetooth, Wap, and Wireless LANs**. McGraw-Hill Professional Book Group, 2000.

HOPKINs, Bruce. **Getting Started with Java and Bluetooth**. Disponível em <http://today.java.net/pub/a/today/2004/07/27/bluetooth.html>. Acesso em 22 de outubro de 2006.

HOPKINS, Bruce; ANTONY, Ranjith. **Bluetooth for Java**. Apress, 2003. 352p.

IBM. **IBM Wireless**. Disponível em <http://www-

128.ibm.com/developerworks/wireless/>. Acesso em 22 de outubro de 2006.

ILYAS, Mohammad. **The Handbook of Ad hoc Wireless Networks**. CRC Press, 2002.

JAVA BLUETOOTH. **Java Bluetooth**. Disponível em <http://www.javabluetooth.com/>. Acesso em 22 de outubro de 2006.

JAVA COMMUNITY PROCESS. **Mobile Information Device Profile for the J2ME Platform Specification**. Disponível em <http://jcp.org/en/jsr/detail?id=37>. Acesso em 22 de outubro de 2006.

______. **Mobile Information Device Profile 2.0 Specification**. Disponível em <http://jcp.org/en/jsr/detail?id=118>. Acesso em 22 de outubro de 2006.

______. **Java APIs for Bluetooth Specification**. Disponível em <http://jcp.org/en/jsr/detail?id=82>. Acesso em 22 de outubro de 2006.

J2ME POLISH. **Devices supporting the Bluetooth API**. Disponível em <http://www.j2mepolish.org/devices/devices-btapi.html>. Acesso em 22 de outubro de 2006.

KJHOLE. **Kjhole.com: Wireless Security Courses** . Disponível em <http://www.kjhole.com/>. Acesso em 22 de outubro de 2006.

KLINGS. **Klings**. Disponível em <http://wireless.klings.org/>. Acesso em 22 de outubro de 2006.

KLINGSHEIM, André N. **J2ME Bluetooth Programming**. 2004. 183f. Dissertação (Mestrado) da University of Bergen, Bergen, 2004.

KUMAR, C. Bala; KLINE, Paul J.; THOMPSON, Timothy J.. **Developing Bluetooth Applications in Java:** Part 1. Disponível em <http://www.commsdesign.com/design_corner/OEG20030611S0026>. Acesso em 22 de outubro de 2006.

______. Developing Bluetooth Applications in Java: Part 2. Disponível em <http://www.commsdesign.com/showArticle.jhtml?articleID=53200223>. Acesso em 22 de outubro de 2006.

______. Bluetooth Application Programming with the Java APIs. Morgan Kaufmann, 2004. 498p.

KUMAR, C. Bala; KLINE, Paul; VAIDYANATHAN, Ranjani. **Extending the Java™ 2 Platform, Micro Edition for Bluetooth Applications**. JavaOne. TS-3092. 2002. Acesso em 22 de outubro de 2006.

JABWT. **JABWT Email Group**. Disponível em <http://groups.yahoo.com/group/JABWT/>. Acesso em 22 de outubro de 2006.

LERG, Andreas. **Wireless Networks:** LAN and Bluetooth. Data Becker, 2002. 160p.

MAHMOUD, Qusay H. **Wireless Application Programming with J2ME and Bluetooth**. Disponível em <http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth1/>. Acesso em 22 de outubro de 2006.

______. **Part II: The Java APIs for Bluetooth Wireless Technology**. Disponível em <http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth2/>. Acesso em 22 de outubro de 2006.

MCDERMOTT-WELLS, Patricia. **What is Bluetooth?**. Potentials, IEEE, 2005. Disponível em <http://ieeexplore.ieee.org/search/freesrchabstract.jsp?arnumber=1368913&isnumber=29958&punumber=45&k2dockey=1368913@ieeejrns&query=%28+%28+bluetooth%3Cin%3Emetadata+%29+%29+%3Cand%3E+%28pyr+%3E%3D+2005+%3Cand%3E+pyr+%3C%3D+2006%29&pos=0>

MILLER, Michael. **Discovering Bluetooth**. Sybex, Incorporated, 2001. 304p.

MOBILE INFO. UPS Uses WiFi & Bluetooth Together To Manage Packages at Shipping Hubs. **MobileInfo**, 22 dez. 2002 Disponível em <http://www.mobileinfo.com/News_2002/Issue47/UPS_WiFi_Bluetooth.htm>. Acesso em 22 de outubro de 2006.

MORROW, Robert. **Bluetooth Operation and Use**. McGraw-Hill Professional, 2002. 576p.

MOTODEV. **MOTODEV**. Disponível em <http://developer.motorola.com>. Acesso em 22 de outubro de 2006.

MUCHOW, John W. **Core J2ME -** Tecnologia & MIDP. 1. ed. São Paulo: Pearson Makron Books, 2004.

MUNDO SEM FIO. Fiat Stilo 2005 terá opção do Kit Connect, 11 abr. 2006. **Mundo sem fio**. Disponível em <http://www.mundosemfio.com.br/news/000057.shtml>. Acesso em 22 de outubro de 2006.

NOPPA. Personal Navigation and Information System for Users of Public Transport. **Noppa**. Disponível em <http://virtual.vtt.fi/noppa/noppaeng.htm>. Acesso em 22 de outubro de 2006.

NOWIRESS. **NoWiress Research Group**. Disponível em <http://www.nowires.org/>. Acesso em 22 de outubro de 2006.

OLIVEIRA, Cesar Rodrigo Campos Leite de. **Controle de Sistema via Bluetooth**. 2005. 59f. Dissertação (Bacharel em Engenharia da Computação) - Faculdade de Engenharia de Sorocaba, Sorocaba, 2005.

ORTIZ, C. Henrique. **Using the Java APIs for Bluetooth Wireless Technology, Part 1 -** API Overview. 2004. Disponível em <http://developers.sun.com/techtopics/mobility/apis/articles/bluetoothintro/>. Acesso em 22 de outubro de 2006.

______. **Using the Java APIs for Bluetooth, Part 2 -** Putting the Core APIs to Work. 2005. Disponível em <http://developers.sun.com/techtopics/mobility/apis/articles/bluetoothcore/index.html>. Acesso em 22 de outubro de 2006.

______. **A Survey of J2ME Today**. Disponível em <http://developers.sun.com/techtopics/mobility/getstart/articles/survey/>. Acesso em 22 de outubro de 2006.

PALO WIRELESS. **PaloWireless – Wireless Resource Center**. Disponível em <http://www.palowireless.com/>. Acesso em 22 de outubro de 2006.

PEDRALHO, André de Souza; NETTO, Antônio Gomes de Araújo; JÚNIOR, Tomaz Nolêto Silva. Bluetooth: da teoria à prática. **Revista Web Mobile**, Grajaú. v. 1, n. 3, p. 16-20, jun./jul. 2005.

PHONEY WORLD. Moto and Oakley - Bluetooth Shades!. **Phoney World**. Disponível em <http://www.phoneyworld.com/news.aspx?news=Moto%20and%20Oakley%20-%20Bluetooth%20Shades! >. Acesso em 22 de outubro de 2006.

PÓVOA, Fábio Mesquita Póvoa. Um jogo J2ME/MIDP multi-usuário usando bluetooth. **Revista Web Mobile**, Grajaú, v. 1, n. 4, p. 44-57, ago./set. 2005.

PRASAD, Ramjee; MUNOZ, Luis. **WLANs and WPANs Towards 4G Wireless**. Artech House Incorporated, 2003. 245p.

PRABHU, C.S.R.; REDDI, Prathap A. **Bluetooth Technology and Its Applications with JAVA and J2ME**. Prentice Hall of India, 2006. 340p.

ROCOCO. **Rococo**. Disponível em <http://rococosoft.com/>. Acesso em 22 de outubro de 2006.

SAMSUNG. Samsung SPD-50P91FHD - The first Bluetooth certified TV. <http://www.newlaunches.com/archives/samsung_spd50p91fhd_the_first_bluetooth_certified_tv.php>. Acesso em 27 de março de 2007.

SIEP, Tom. **An IEEE Guide: How to Find What You Need in the Bluetooth Spec**. IEEE STANDARD OFFICE, 2000. 150p.

SONY ERICSSON. **Mobile Developer Support – SoyEricsson Developer World**. Disponível em <http://sonyericsson.com/developer/>. Acesso em 22 de outubro de 2006.

STURMAN, Charles e BRAY, Jennifer. **Bluetooth 1.1: Connect Without Cables, Second Edition**. Prentice Hall PTR, 2001. 622p.

SUN MICROSYSTEMS. **Java ME – Micro App Development Made Easy**. Sun Microsystems, Inc, 2006a. Disponível em <http://java.sun.com/javame>. Acesso em 22 de outubro de 2006.

SUN MICROSYSTEMS. **Java SE – Overview at a Glance**. Sun Microsystems, Inc, 2006b. Disponível em <http://java.sun.com/javase>. Acesso em 22 de outubro de 2006.

SUN MICROSYSTEMS. **Mobile Information Device Profile**. Sun Microsystems, Inc, 2006c. Disponível em <http://java.sun.com/products/midp/>. Acesso em 22 de outubro de 2006.

SUN MICROSYSTEMS. **Connected Device Configuration**. Sun Microsystems, Inc, 2006d. Disponível em <http://java.sun.com/products/cdc/>. Acesso em 22 de outubro de 2006.

SUN MICROSYSTEMS. **Connected Limited Device Configuration**. Sun Microsystems, Inc, 2006e. Disponível em <http://java.sun.com/products/cldc/>. Acesso em 22 de outubro de 2006.

SYMBIAN DEVELOPER. **Symbian Developer**. Disponível em <http://www.symbian.com/developer>. Acesso em 22 de outubro de 2006.

TRANTER, William H. **Wireless Personal Communications : Bluetooth and Other Technologies**. Kluwer Academic Publishers, 2000.

WANG, Alf Inge; NORUM, Michael Sars; LUND, Carl-Henrik Wolf. **Issues related to Development of Wireless Peer-to-Peer Games in J2ME.** 2006. Noruega, 2006.

WIRELESS DEVELOPER. Disponível em <http://www.wirelessdevnet.com/channels/bluetooth/>. Acesso em 22 de outubro de 2006.

ANEXO A - Documentos

A.1 Artigo do Projeto Marge

Marge: Um Framework para Desenvolvimento de Aplicações Java que utilizam a Tecnologia Bluetooth

Bruno Cavaler Ghisi, Lucas Bortolaso Torri, Frank Siqueira

Departamento de Informática e Estatística – Universidade Federal de Santa Catarina
CEP 88040-900 - Campus Universitário Cx.P. 476 – Florianópolis – SC – Brasil

{joca,lucastorri,frank}@inf.ufsc.br

Resumo. *Bluetooth é uma tecnologia sem fio que tornou-se recentemente o padrão de fato de intercomunicação para conectar dispositivos a uma curta distância. Ganhou bastante espaço e repercussão em grande parte dos celulares atuais, assim como em notebooks, PDAs e outros dispositivos. A linguagem de programação Java, além de hoje ser uma das principais plataformas de desenvolvimento de aplicações para computadores, tornou-se a linguagem dominante no mercado de aparelhos celulares, devido às diversas vantagens que a mesma oferece. Java possui uma especificação para a incorporação da tecnologia Bluetooth em sua plataforma, definida pela JSR 82 (Java APIs for Bluetooth), que por sua vez apresenta um difícil aprendizado para o desenvolvedor. Esta limitação levou ao desenvolvimento do framework Marge, apresentado neste artigo, que é um projeto de código livre disponível no portal java.net. Esse framework tem o intuito de facilitar o desenvolvimento de aplicações Java que façam o uso de Bluetooth, podendo ser usado tanto na plataforma Java ME, quanto na Java SE. Além do framework, o artigo apresenta também algumas aplicações de exemplo que demonstram o uso do framework Marge para criação de aplicações que utilizam Bluetooth para comunicação.*

Palavras-chave: Bluetooth. Java. JSR 82. Marge.

Abstract. *Bluetooth is a wireless technology which became in recent years a de facto intercommunication standard for connecting devices in a short distance. This technology is available in a large share of the current mobile phones and also in notebooks, PDAs and other devices. The Java programming language, which is today one of the main platforms for computer application development, became the dominant language in the mobile phone market, due to the many advantages which it offers. Java has a specification for incorporating the Bluetooth technology in its platform, defined by the JSR 82 (Java APIs for Bluetooth), which presents a steep learning curve for developers. This limitation lead to the development of the Marge framework, presented in this paper, which is an open source project hosted on java.net. This framework intends to facilitate the development of Java applications that use Bluetooth, being able to be used in Java ME platform, as well as in Java*

SE. Besides the framework, some sample applications were also created, in order to demonstrate the use of the Marge framework for developing applications that use Bluetooth for communication.

Key-words: Bluetooth. Java. JSR 82. Marge.

1. Introdução

As tecnologias sem fio têm se tornado muito populares devido a sua praticidade e comodidade. Uma grande quantidade delas vem surgindo ao longo do tempo para suprir diferentes necessidades. Uma delas, a tecnologia Bluetooth, criada com o intuito de eliminar a necessidade de utilização de cabos elétricos para conexão de equipamentos, hoje está presente em diversos aparelhos, tais como: impressoras, *notebooks*, câmeras, fones de ouvidos, televisões, PDAs e principalmente em telefones celulares.

Por outro lado, a tecnologia Java disponibiliza uma linguagem poderosa que permite suporte a programação em diversos contextos distintos, fator que contribuiu para que essa linguagem, assim como Bluetooth, dominasse o mercado de dispositivos móveis.

Para permitir o suporte da tecnologia Bluetooth em Java, foi definida a especificação JSR 82 (*Java APIs for Bluetooth*). Porém, essa especificação possui um difícil aprendizado devido a sua complexidade. Com o propósito de facilitar o desenvolvimento de aplicações Java que façam uso de Bluetooth, o presente artigo descreve o Projeto Marge, que tem como um de seus objetivos suprir essa necessidade por meio da disponibilização de um *framework* para desenvolvimento de aplicações em Java que utilizem a tecnologia Bluetooth para comunicação.

O restante do presente artigo está organizado da seguinte maneira: o capítulo 2 descreve a tecnologia Bluetooth; a plataforma Java é descrita no capítulo 3, com especial atenção para a sua versão para dispositivos móveis, o Java ME; o capítulo 4 descreve o *framework* Marge e alguns programas que demonstram a sua utilização; por fim, são apresentadas no capítulo 5 as conclusões e perspectivas de evolução do trabalho descrito neste artigo.

2. Tecnologia Bluetooth

Bluetooth é uma especificação para redes pessoais sem fio, criada em 1994 pela empresa Ericsson Mobile Communication com o objetivo de interconectar diferentes dispositivos, oferecendo baixo consumo de energia, pequeno custo de incorporação a outros projetos, junto a uma taxa considerável de transferência de dados.

No começo de 1997, a Ericsson Mobile Communication aproximou-se de outros fabricantes de dispositivos portáteis tentando promover o uso desta tecnologia, fazendo com que outras empresas também trabalhassem no desenvolvimento da mesma. Tal estratégia foi adotada pois, para o Bluetooth ser bem sucedido, seria necessário que uma quantidade considerável de aparelhos utiliza esse mesmo padrão para comunicação, através do suporte de um grande conjunto de empresas.

Sendo assim, em fevereiro de 1998, foi criado o Bluetooth SIG (S*pecial Interest Group*) em parceria com a Nokia, IBM, Toshiba e Intel, reunindo então empresas da área de telefonia móvel, computadores portáteis e processadores. O consórcio foi anunciado para o mundo em maio daquele mesmo ano e, no decorrer dos tempos, teve um grande crescimento,

contando com a adesão de outras empresas renomadas como Motorola, Dell, 3Com, Compaq, Qualcomm, Samsung, Siemens, Symbian e outras. Atualmente o Bluetooth SIG já conta com mais de 8000 empresas participantes [BLUETOOTH SIG, 2007].

O nome Bluetooth surgiu em homenagem a um rei viking do século X chamado de Danish King Harald Blatand - ou Harald Bluetooth, traduzindo para o inglês [BLUETOOTH TECHNOLOGY, 2007]. Esse rei unificou partes da Noruega, Suécia e Dinamarca. O SIG optou por esse nome pelo fato de um dos objetivos da tecnologia ser a unificação de diferentes indústrias através de uma mesma tecnologia de comunicação.

No final do ano passado, o Bluetooth SIG (2007), anunciou que as vendas de dispositivos Bluetooth - em sua maioria celulares - tinham chegado a um marco de 1 bilhão, o que representa mais que o número de computadores pessoais no mundo. Recentemente, em março deste ano, também anunciou que a tecnologia continuava a crescer em popularidade, informação apontada por uma pesquisa que anunciava que 81% dos entrevistados já conheciam a tecnologia Bluetooth.

2.1. Informações Técnicas

Bluetooth utiliza-se de radiofreqüência para transmissão de dados e opera na faixa ISM (*Industrial, Scientific, Medical*) - banda essa utilizada por instrumentos com sinal eletromagnético nas áreas industrial, científica e médica - centrada em 2,45 GHz.

A tecnologia pode ser dividida em três classes: 1, 2 e 3. Cada classe é destinada a um grupo de aparelhos, havendo variação do alcance - respectivamente 100, 10 e 1 metro - e do consumo de energia para cada uma, proporcional ao aumento do alcance.

A tecnologia também se utiliza de saltos de freqüência (*Frequency-hopping*) para diminuir as chances de interferência na comunicação entre os dispositivos. Essa técnica divide a banda de comunicação em partes menores e faz saltos pseudo-randômicos entre essas freqüências, evitando o sobre uso de uma mesma faixa por diferentes canais de comunicação.

2.2. Piconet/Scatternet

Uma rede *piconet* é a forma básica de uma rede Bluetooth e consiste em um dispositivo mestre (*master*) e de um até sete escravos (*slaves*) conectados a ele. Essas conexões criam uma rede em forma de estrela, com o mestre no nodo central.

O mestre é o dispositivo que inicia uma conexão Bluetooth, e é o responsável por sincronizar a faixa de freqüência usada por ele e pelos demais dispositivos a ele conectados, assim como definir tempos de transmissão de dados para os escravos. Toda comunicação tem de passar primeiramente pelo mestre, fazendo com que os escravos não possam se comunicar diretamente entre eles.

Duas ou mais *piconets* podem se interconectar, criando então uma *scatternet*. Essa conexão é feita por um nodo intermediário, que compartilha o tempo seguindo a faixa de freqüência de cada *piconet* por vez. Isso reduz a fatia de tempo disponível para transferência de dados entre o nodo intermediário e o mestre, dividindo-a, pelo menos, pela metade. A figura 1 detalha uma rede *scattenet* formada por 12 *piconets*.

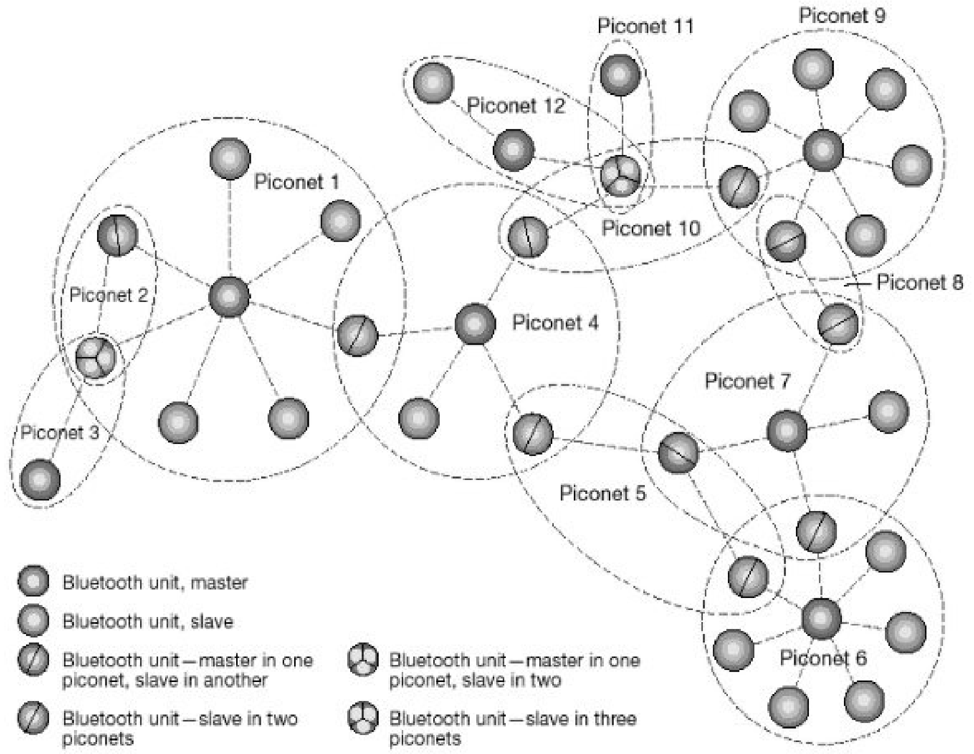


Figura 1. Piconet e Scatternet [FRODIGH, 2002].

2.3. Serviços Bluetooth

Dispositivos Bluetooth possuem a habilidade de prover serviços. Diferentes dispositivos Bluetooth podem oferecer diferentes serviços. Um dispositivo Bluetooth, após ter encontrado outro dispositivo remoto que esteja próximo, poderá fazer uma pesquisa pelos serviços disponíveis nesse dispositivo remoto ou apenas por algum serviço específico. A busca por dispositivos utiliza o protocolo SDP (*Service Discovery Protocol*).

Um identificador único - *Universally Unique Identifier* (UUID) - é usado para identificar os serviços, protocolos, perfis e demais informações do dispositivo. Um UUID é um identificador de até 128 bits que garante sua unicidade em tempo e espaço. A tecnologia Bluetooth utiliza diferentes variantes de UUIDs, com identificadores de diferentes tamanhos. Isso é feito com o intuito de reduzir eventuais armazenamento e transferências desnecessárias de valores de UUIDs de 128 bits. Existe uma gama de UUIDs pequenos pré-alocados para serviços freqüentemente usados, protocolos e perfis.

2.4. Protocolos

Como o intuito da especificação Bluetooth é permitir que dispositivos de fabricantes distintos interconectem-se de forma compatível e interoperável, não é apenas suficiente a existência do sistema de rádio em hardware, havendo também a necessidade de uma complexa pilha de protocolos em software que garantam todo esse funcionamento [BLUETOOTH TECHNOLOGY, 2006], conforme mostrado na Figura 2.

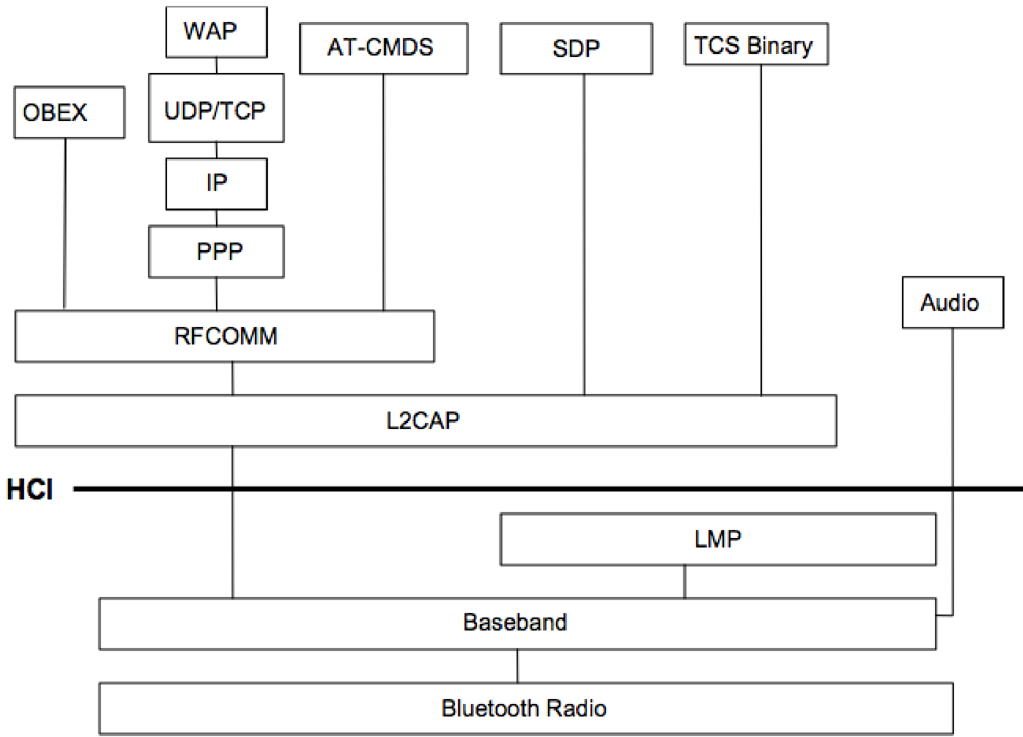


Figura 2. Pilha de Protocolos Bluetooth [JAVA APIS FOR BLUETOOTH, 2007].

Dentre os protocolos de comunicação disponíveis na tecnologia Bluetooth, três se destacam:

- L2CAP (*Logical Link Control and Adaptation Protocol*): é o protocolo básico para comunicação via Bluetooth. Situa-se logo acima da camada de hardware e provê funções importantes como multiplexação para as camadas de comunicação mais altas e segmentação e montagem dos pacotes enviados.
- RFCOMM (*Radio Frequency Communication*): fornece emulação de portas seriais RS232 através do protocolo L2CAP, provendo múltiplas conexões ao dispositivo num mesmo intervalo de tempo.
- OBEX (*Object Exchange*): é o mais complexo dos três, consistindo em requisições e respostas como em um esquema cliente/servidor, possibilitando a troca de objetos em formatos binários entre dispositivos. Foi incorporado da tecnologia de infravermelho (IrDA), com o objetivo de manter a interoperabilidade entre as tecnologias.

3. Java

Java é uma tecnologia criada na década de 90 pela empresa Sun Microsystems, onde seu principal componente é uma linguagem de programação que ficou conhecida pelos desenvolvedores por Java.

Diferentemente das linguagens convencionais da época, que passavam por um processo de compilação afim de obter código nativo, a linguagem Java é transformada para um código intermediário, chamado de *bytecode*, que é executado por uma máquina virtual, a JVM (*Java Virtual Machine*). Essa característica do código ser interpretado em tempo de execução proporciona independência de plataforma à linguagem, consagrando uma das

maiores características da mesma: a portabilidade.

Para cobrir seu uso em diferentes tipos de sistemas, a tecnologia Java foi dividida em três plataformas: Java ME (*Java Platform, Micro Edition*); Java SE (J*ava Platform, Standard Edition*) e Java EE (*Java Platform, Enterprise Edition*). Neste trabalho focamos nas duas primeiras, por estas serem destinadas a equipamentos que tipicamente possuem suporte a Bluetooth.

3.1. Java SE

Java SE é a plataforma para desenvolvimento Java de aplicações para desktop e servidores, usada como base para as demais plataformas [JAVA SE, 2007]. Além da JVM e do compilador, é formada por um conjunto de bibliotecas – a Java API (*Application Programming Interface*).

3.2. Java ME

Java ME provê um ambiente destinado à execução de aplicações Java em dispositivos móveis e embarcados. Essa categoria engloba celulares, PDAs, *set-top boxes* – conversores usados em televisão digital - e outros aparelhos com características de baixa disponibilidade de recursos [JAVA ME, 2007].

Como a quantidade de recursos disponíveis pode variar significamente conforme os dispositivos existentes, foram criados os conceitos de configurações, perfis e pacotes adicionais para adequar a tecnologia a cada um deles.

3.2.1. Configurações

Uma configuração define as funcionalidades mínimas para um grupo de dispositivos com características semelhantes. Sendo assim, mais especificamente, ela define as características quanto à JVM e o conjunto de classes derivadas da plataforma Java SE.

Atualmente, o Java ME divide as configurações em dois tipos: CLDC (*Connected Limited Device Configuration*), para configurações mais limitadas, como telefones celulares; e CDC (*Connected Device Configuration*) para configurações com mais recursos, por exemplo os PDAs. A CLDC é um subconjunto da CDC.

3.2.2. GCF (*Generic Connection Framework*)

Durante o desenvolvimento do Java ME, as APIs *java.io* e *java.net* do Java SE foram consideradas grandes demais para serem usadas nos dispositivos móveis, então o GCF foi especificado para substituí-las.

Como o nome sugere, o *framework* de conexão genérica é usado par fazer diversos tipos de conexões, tais como HTTP, *streams*, datagramas e outras. Ele foi especificado para uso em CLDC, entretanto, pelo fato de ser extremamente extensível, atualmente já é possível usá-lo com CDC através de perfis que o implementam, assim como em Java SE, através da especificação JSR 197.

3.2.3. Perfis

Um perfil fornece as bibliotecas necessárias para que o desenvolvedor possa escrever aplicativos para um determinado tipo de dispositivo.

Existem diferentes perfis que suprem necessidades distintas nas duas configurações (CDC e CLDC), sendo o mais comum deles o MIDP (*Mobile Information Device Profile*).

O perfil MIDP é voltado especificamente para dispositivos portáteis, como celulares e PDAs (de baixo desempenho). Este perfil proporciona funcionalidades como gerenciamento do ciclo de vida de uma aplicação, componentes de interface gráfica e persistência dos dados.

3.2.4. Pacotes Opcionais

A plataforma Java ME pode ser estendida pela combinação de vários pacotes opcionais, conforme ilustrado na Figura 3. Esses pacotes opcionais, que não necessitam necessariamente ser disponibilizados pelo fabricante, são criados para atingir requisitos específicos de mercado, oferecendo APIs padrões para tecnologias existentes e emergentes, tais como o Bluetooth.

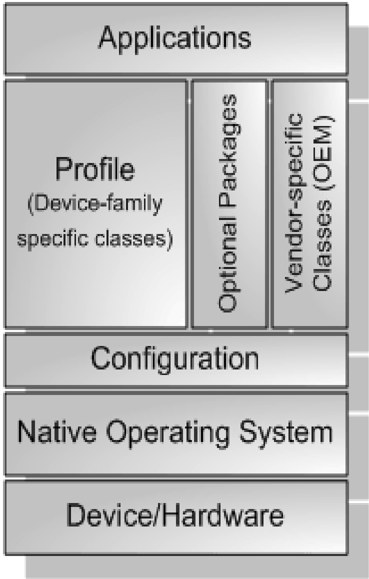


Figura 3. Estrutura Java ME com configurações, perfis e pacotes opcionais.

3.3. JSR 82 (*Java APIs For Bluetooth*)

A JSR 82 é um pacote opcional que define o uso de Bluetooth dentro da tecnologia Java, utilizando-se do GCF para disponibilizar a comunicação. A especificação está dividida em dois pacotes principais: *javax.bluetooth* e *javax.obex*.

O pacote *javax.bluetooth* é o núcleo dessa especificação. Provê uma API para busca de dispositivos e procura de serviços, além de fornecer funcionalidades para comunicação através dos protocolos L2CAP e RFCOMM.

O pacote *javax.obex* provê uma API para transferência de objetos via OBEX. Por não

ser obrigatório na especificação, atualmente poucos dispositivos implementam esse pacote.

Inicialmente, a JSR 82 foi desenvolvida para o uso da configuração CLDC com o perfil MIDP, devido a sua dependência do GCF. Porém, como existem implementações do GCF para Java SE, é possível usar implementações da JSR 82 nessa plataforma. Algumas dessas implementações são: Avetana [AVETANA, 2007], Bluecove [BLUECOVE, 2007] e Rococo [ROCOCO, 2007].

4. Projeto Marge

O projeto Marge é um projeto de código aberto nascido no Brasil, registrado sob a licensa LGPL (*Lesser General Public License*), sendo formado por colaboradores de várias partes do mundo.

Foi criado com o intuito de facilitar o desenvolvimento de aplicações em Java que façam uso do Bluetooth, como por exemplo: jogos, controles remoto, aplicações de marketing, soluções para automação residencial, entre outras.

O nome deriva da personagem Marge Simpson, do desenho animado 'Os Simpsons'[6]. Essa personagem caracteriza-se por possuir um cabelo alto de cor azul com formato cilíndrico, o que no pensamento dos autores faz alusão a uma rede Bluetooth.

Figura 4. Logotipo do Projeto Marge.

4.1. Estrutura

O Projeto Marge está hospedado no portal *java.net* (http://marge.dev.java.net), sendo um projeto reconhecido pela comunidade *Mobile & Embedded* [MOBILE AND EMBEDDED, 2007], responsável pela área de projetos relacionados à tecnologia Java ME.

O projeto utiliza diversas ferramentas disponibilizadas pelo portal, tais como o *Subversion* para controle de versões do código, listas de e-mail, sistema de *bug tracking* para planejamento e organização das tarefas a serem desenvolvidas e *wiki* para documentação.

Todo código é escrito em inglês, para facilitar o entendimento de novos desenvolvedores, já que a mesma é considerada como língua universal. Esse código está basicamente dividido em dois segmentos: *marge-core* e *marge-demos*.

4.2. *marge-core*

[6] Do original em inglês 'The Simpsons' © 20th Century Fox Television. Criado por Matt Groening.

O *marge-core* e é um *framework* criado sobre a JSR 82, conforme mostra a Figura 5, para facilitar o desenvolvimento de aplicações que façam uso dessa API, sendo compatível tanto com Java ME, quanto com Java SE. Atualmente está na sua segunda versão estável, a 0.4.0.

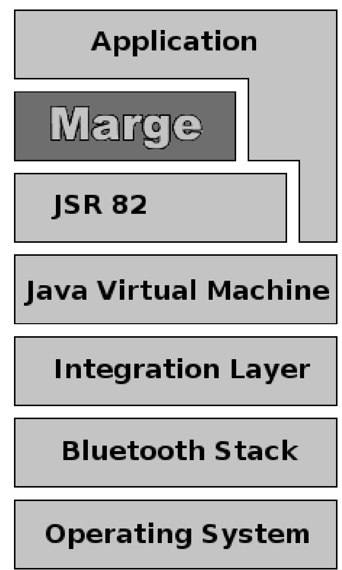


Figura 5. Arquitetura do *marge-core*.

Esse *framework* é responsável por abstrair o difícil aprendizado da JSR 82, sendo possível desenvolver aplicações que usem Bluetooth de forma muito mais rápida, em um nível de programação mais alto.

Dentre as vantagens que a versão atual proporciona estão a facilidade no processo de busca por dispositivos e pesquisa por serviços, conexão entre clientes e servidor, troca de mensagens nos protocolos L2CAP e RFCOMM, suporte a diferentes configurações, entre outras, sendo que cada uma dessas funcionalidades pode ser utilizada separadamente. Atualmente o *framework* está dividido em 4 pacotes e possui 19 classes. A estrutura básica de pacotes está listada abaixo:

- *net.dev.marge.communication* - Apresenta as classes relativas ao processo de conexão e comunicação, utilizando os protocolos L2CAP e RFCOMM.
- *net.dev.marge.entity* - Contém as classes das entidades que representam cliente e servidor, junto com as configurações necessárias para instanciação das mesmas.
- *net.dev.marge.factory* - Possui fábricas que instanciam as entidades cliente e servidor, recebendo as configurações como parâmetro.
- *net.dev.marge.inquiry* - Disponibiliza classes que facilitam a busca por dispositivos e pesquisa pelos serviços nos dispositivos.

O processo de estabelecer uma conexão Bluetooth entre dispositivos é feito utilizando o modelo cliente/servidor. Clientes e servidores são representados, respectivamente, pelas classes *ClientDevice* e *ServerDevice*, contidas no pacote *net.dev.marge.entity*. Essas representações são utilizadas para fazer o envio de mensagens para a rede Bluetooth a ser formada.

A Figura 6 mostra um exemplo de utilização do Marge para criação de um servidor

Bluetooth. Inicialmente é necessária a criação de uma entidade servidor, obtida através das fábricas presentes no pacote *net.dev.marge.factory.* São elas*:* *RFCOMMCommunicationFactory* para servidores que utilizem o protocolo RFCOMM e *L2CAPCommunicationFactory* quando utilizado o protocolo L2CAP. Para essa criação, além de um conjunto de configurações, é passada também uma instância da interface *ConnectionListener*, que será notificada a cada nova conexão estabelecida com um cliente.

```
// Criando uma fábrica utilizando o protocolo RFCOMM
CommunicationFactory factory = new RFCOMMCommunicationFactory();
// Criando a configuração do servidor
ServerConfiguration sconf = new ServerConfiguration(new
CommunicationListenerImpl());
// Instanciando o servidor via fábrica
factory.waitClients(sconf, new ConnectionListenerImpl());
// Iniciando a escuta no servidor
serverDevice.startListening();
// Servidor enviando uma mensagem
serverDevice.send("Test message".getBytes());
// Fechando a conexão
serverDevice.close();
```

Figura 6 – Exemplo de código do Servidor.

A Figura 7 mostra um exemplo de código de um cliente Bluetooth construído utilizando as classes fornecidas pelo *framework*. Para se estabelecer a conexão entre as entidades, o cliente deve buscar pelo servidor ao qual deseja se conectar e escolher um dos serviços disponíveis. Essa pesquisa por dispositivos e busca por serviços é feita, respectivamente, através das classes *DeviceDiscoverer* e *ServiceDiscoverer*, ambas contidas no pacote *net.dev.marge.inquiry.*

Após encontrado o dispositivo que oferece o serviço desejado, cria-se então, pela mesma fábrica usada para criação do servidor, um cliente que se conectará a esse serviço. A partir desse momento está estabelecida a conexão, que será controlada internamente pelas classes contidas no pacote *net.dev.marge.communication.*

```
// Iniciando a busca por dispositivos
DeviceDiscoverer.getInstance().startInquiry(DiscoveryAgent.GIAC, new
InquiryListenerImpl());
// Iniciando a busca por serviços em um determinado dispositivo encontrado
ServiceDiscoverer.getInstance().startSearch(uuidArray, remoteDeviceFound, new
ServiceSearchListenerImpl());
// Criando uma fábrica utilizando o protocolo RFCOMM
CommunicationFactory factory = new RFCOMMCommunicationFactory();
// Criando a configuração do cliente
ClientConfiguration cconfig = new ClientConfiguration(serviceRecord, new
CommunicationListenerImpl());
// Instanciando o cliente via fábrica
```

```
ClientDevice clientDevice = factory.connectToServer(cconf);
// Iniciando a escuta no cliente
clientDevice.startListening();
// Cliente enviando uma mensagem
clientDevice.send("Test message".getBytes());
// Fechando a conexão
clientDevice.close();
```

Figura 7 – Exemplo de código do Cliente.

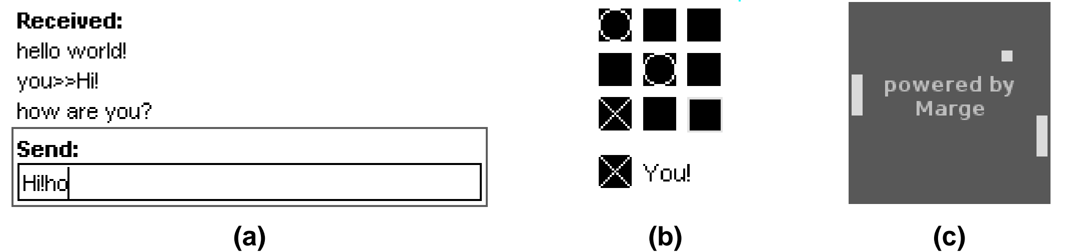
As configurações *ClientConfiguration* e *ServiceConfiguration*, pertencentes ao pacote *net.dev.marge.entity.configuration*, possuem atributos responsáveis por determinar certas configurações dos clientes e servidores. Essas configurações são enviadas às fábricas e utilizadas no processo de instanciação das entidades.

Dentre algumas configurações que podem ser atribuídas, estão: uma instância da interface *net.dev.marge.communication.CommunicationListener* que será notificada da chegada de novas mensagens, o intervalo de tempo para leitura de chegada de novas mensagens, entre outras configurações específicas para cada uma das entidades.

4.3. *marge-demos*

O *marge-demos* é o conjunto de aplicações que utilizam e apresentam o uso de Bluetooth através do Projeto Marge. Atualmente ele é composto pelas seguintes aplicações:

- *Blue Chat* - Chat em Java ME que comporta até 8 pessoas/dispositivos conectadas e conversando, cuja interface gráfica é mostrada na Figura 8(a). É criado um servidor que pode utilizar o protocolo RFCOMM ou L2CAP e os demais dispositivos conectam-se à ele, abrindo o Chat entre os dispositivos conectados.
- *Tic Tac Tooth* - Jogo da velha em Java ME entre dois jogadores, ilustrado na Figura 8(b). Após estabelecida a conexão entre os aparelhos, os jogadores fazem suas jogadas e o processamento das mesmas é feito de forma distribuída.
- *Blue Pong* - Jogo em Java ME, no estilo do clássico Pong, entre dois jogadores, onde o objetivo é fazer uma bola passar da linha vertical do seu adversário. Sua interface gráfica é mostrada na Figura 8(c).
- *OOo Impress Slideshow Controller* - Passador de slides para o *Open Office Impress* que pode ser usado através do celular. Possui um cliente Java ME e um servidor Java SE que controla o *Open Office* utilizando a API UNO (*Universal Network Objects*), disponibilizada pelo mesmo. Eles se comunicam trocando arquivos no formato XML (*Extensible Markup Language*).
- *Presence Notifier* – Interação entre software e *hardware* que permite notificar a presença de um dispositivo Bluetooth presente na área, através do acendimento de um LED (*light-emitting diode*). Consiste de um *software* em Java SE, que usa o *marge-core*, fazendo chamadas seriais a um dispositivo externo que possui um programa escrito em linguagem C.



(a) (b) (c)

Figura 8 – Interfaces gráficas dos demos Blue Chat (a), Tic Tac Tooth (b) e Blue Pong (c).

5. Conclusão

Bluetooth é uma tecnologia sem fio muito utilizada para conectar dispositivos a curta distância, que vem ganhando inúmeros adeptos devido a suas qualidades. Além disso, ela passou por um grande amadurecimento ao longo de sua criação, tornando-se um padrão de fato para comunicação entre dispositivos móveis, contando hoje com o apoio de grandes empresas.

Em paralelo a isso, também é notável o crescimento da tecnologia Java em diversas áreas, inclusive em dispositivos móveis e embarcados, onde já domina esse mercado.

A especificação JSR 82 é caracterizada por possuir um difícil aprendizado. O Projeto Marge surgiu com o intuito de facilitar o desenvolvimento, em Java, de aplicações que usem Bluetooth, visando suprir essa dificuldade. Assim, é possível explorar esse forte padrão de comunicação através de uma linguagem robusta, em um mercado que se caracteriza pela geração de novos produtos e serviços sem fio.

Por se tratar de um projeto em constante evolução, o Marge conta ainda com várias idéias a serem desenvolvidas. Entre elas estão: aumentar o nível de abstração do *framework*; implementação do protocolo de comunicação OBEX; criação de extensões para Java ME (incluindo uma específica para jogos) e para Java SE; melhorias nos testes de unidade; melhoria e localização da documentação; internacionalização e localização das aplicações de exemplo, assim como a criação de novas aplicações.

Referências

AVETANA. avetanaBluetooth JSR82 implementation for Windows, MacOS X and Linux Systems. Disponível em <http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.eng.xml>. Acesso em 17 de Junho de 2007.

BLUECOVE. BlueCove - BlueCove examples projects. Disponível em <http://bluecove.sourceforge.net/>. Acesso em 17 de Junho de 2007.

BLUETOOTH SIG. Bluetooth SIG. Disponível em <https://www.bluetooth.com/>. Acesso em 17 de Junho de 2007.

FRODIGH, M., Johansson, P. and Larsson, P. Wireless ad hoc networking - The art of networking without a network. Ericsson Review No. 4, 2002.

JAVA APIS FOR BLUETOOTH. The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 82. Disponível em <http://jcp.org/en/jsr/detail?id=82>. Acesso em 17 de Junho de 2007.

JAVA ME. Java ME – Micro App Development Made Easy. Disponível em

<http://java.sun.com/javame>. Acesso em 17 de Junho de 2007.

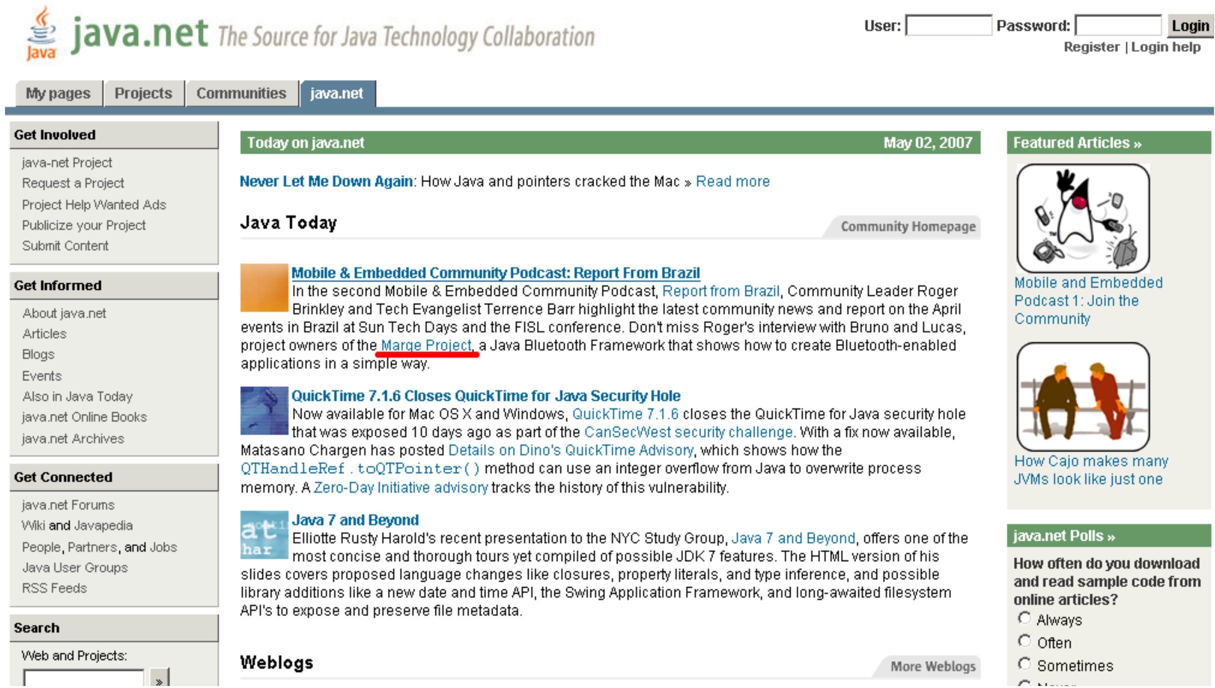
JAVA SE. Java SE – Overview – at a Glance. Disponível em <http://java.sun.com/javase>. Acesso em 17 de Junho de 2007.

MOBILE AND EMBEDDED. community.java.net - Mobile & Embedded Community. Disponível em <http://mobileandembedded.org>. Acesso em 17 de Junho de 2007.

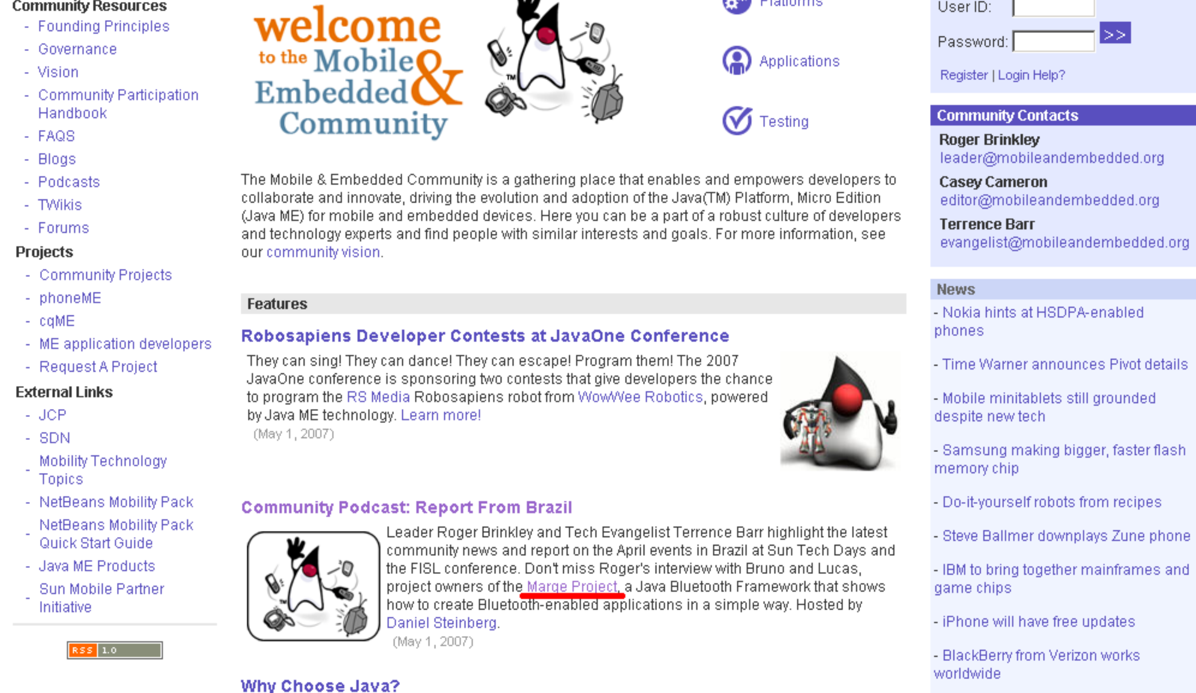
ROCOCO. Java/Bluetooth. Disponível em <http://www.rococosoft.com/java.html>. Acesso em 17 de Junho de 2007.

ANEXO B - Referências ao projeto

B.1 Destaque em vermelho sobre a entrevista do projeto Marge, realizada durante o FISL 8.0, na página principal do java.net (http://mobileandembedded.org), que foi postada no dia 2 de Maio de 2007

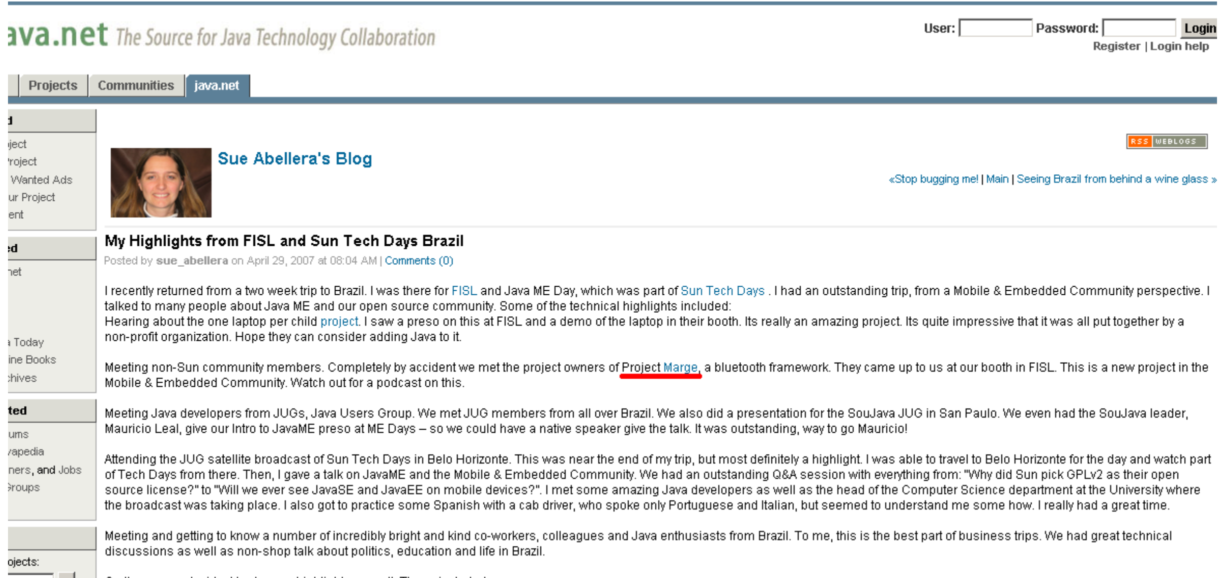


B.2 Destaque em vermelho sobre a entrevista do projeto Marge, realizada durante o FISL 8.0, na página da comunidade Mobile & Embedded do java.net (http://mobileandembedded.org), que foi postada no dia 1 de Maio de 2007



B.3 Destaque em vermelho na página da entrevista do Marge, realizada durante o FISL 8.0 (http://today.java.net/pub/a/today/2007/05/01/mobileandembedded-podcast2.html), que foi postada no java.net dia 1 de Maio de 2007

B.4 Destaque em vermelho do *post* sobre o Marge da Sue Abellera (http://weblogs.java.net/blog/sue_abellera/archive/2007/04/my_highlights_f_1.html), ex-gerente do desenvolvimento da máquina virtual Java ME, no dia 29 de Abril, em seu blog no java.net



ava.net The Source for Java Technology Collaboration

User: Password: Login
Register | Login help

Projects | Communities | java.net

Sue Abellera's Blog

«Stop bugging me! | Main | Seeing Brazil from behind a wine glass »

My Highlights from FISL and Sun Tech Days Brazil

Posted by **sue_abellera** on April 29, 2007 at 08:04 AM | Comments (0)

I recently returned from a two week trip to Brazil. I was there for FISL and Java ME Day, which was part of Sun Tech Days . I had an outstanding trip, from a Mobile & Embedded Community perspective. I talked to many people about Java ME and our open source community. Some of the technical highlights included:
Hearing about the one laptop per child project. I saw a preso on this at FISL and a demo of the laptop in their booth. Its really an amazing project. Its quite impressive that it was all put together by a non-profit organization. Hope they can consider adding Java to it.

Meeting non-Sun community members. Completely by accident we met the project owners of Project Marge, a bluetooth framework. They came up to us at our booth in FISL. This is a new project in the Mobile & Embedded Community. Watch out for a podcast on this.

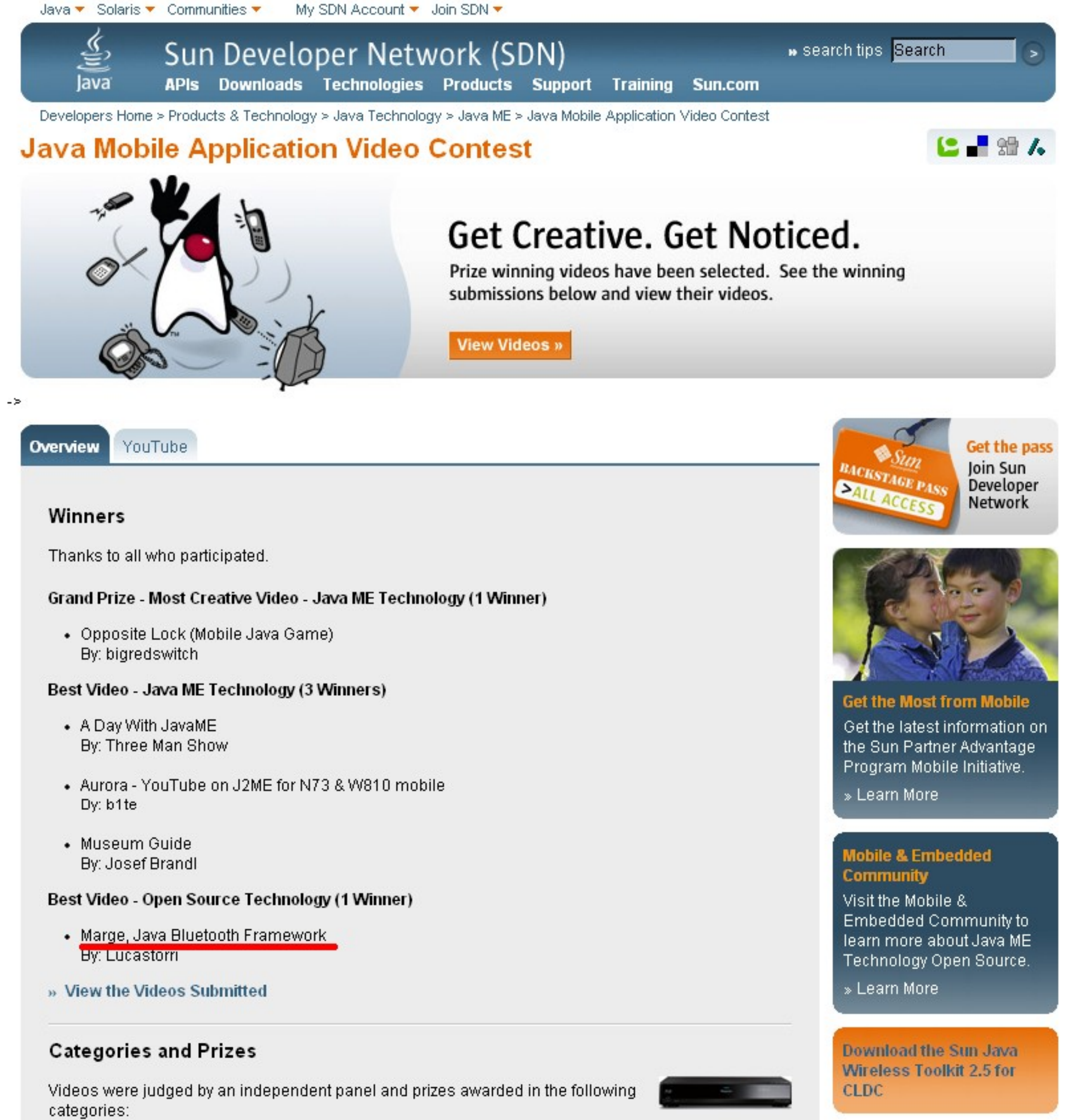
Meeting Java developers from JUGs, Java Users Group. We met JUG members from all over Brazil. We also did a presentation for the SouJava JUG in San Paulo. We even had the SouJava leader, Mauricio Leal, give our Intro to JavaME preso at ME Days – so we could have a native speaker give the talk. It was outstanding, way to go Mauricio!

Attending the JUG satellite broadcast of Sun Tech Days in Belo Horizonte. This was near the end of my trip, but most definitely a highlight. I was able to travel to Belo Horizonte for the day and watch part of Tech Days from there. Then, I gave a talk on JavaME and the Mobile & Embedded Community. We had an outstanding Q&A session with everything from: "Why did Sun pick GPLv2 as their open source license?" to "Will we ever see JavaSE and JavaEE on mobile devices?". I met some amazing Java developers as well as the head of the Computer Science department at the University where the broadcast was taking place. I also got to practice some Spanish with a cab driver, who spoke only Portuguese and Italian, but seemed to understand me some how. I really had a great time.

Meeting and getting to know a number of incredibly bright and kind co-workers, colleagues and Java enthusiasts from Brazil. To me, this is the best part of business trips. We had great technical discussions as well as non-shop talk about politics, education and life in Brazil.

B.5 Destaque em vermelho sobre o *post* do Marge, no dia 23 de Abril, no blog do Neto Marin (http://netomarin.blogspot.com/), um dos criadores do JME Brasil (http://www.jmebrasil.org)

B.6 Destaque em vermelho do projeto Marge que venceu a categoria "Best Video – Open Source Techonology" do concurso Java Mobile Application Video Contest (http://java.sun.com/javame/contest/index.jsp) promovido pela comunidade Mobile & Embedded do java.net, no dia 10 e Maio



B.7 Destaque em vermelho para a entrevista do projeto Marge no blog da empresa irlandesa, que atua nessa área de Bluetooth e Java, Rococo (http://www.rococosoft.com/weblog/archives/2007/05/cool_projects_using_jsr82_1_ma.html), no dia 14 de Maio

Weblog

▶ Recent Entries

Cool Projects using JSR82 [1] : Marge

JSR82 in your project? Let us know!

Bluetooth Gaming

Bluetooth Marketing : Wising Up

Me want (more Bluetooth Stereo Headset Lust)

Some Bluetooth Style

Dangers of Bluetooth

Installed base: 1 Billion; rate per week : 13 million - Bluetooth roll continues

Weblog

Cool Projects using JSR82 [1] : Marge

We sent the call out to JSR82 Java/Bluetooth developers everywhere - tell us about your project! And the replies have started to flow in. So today is the first in a regular series of short articles about projects (academic, hobbyist, commercial, whatever) that use Java/Bluetooth technology and JSR82 specifically in some way. We're structuring these as an interview-style Q&A with the developers themselves, so you hear it from the horse's mouth.

So without further ado, please welcome: The Marge Project!

The brainchild of Bruno Ghisi and Lucas Torri, Marge is about making Java Bluetooth application development even simpler for developers. As you may know, the JSR82 APIs are quite low-level, and tend to slavishly follow the underlying Bluetooth Stack profiles as defined in the Bluetooth Standard. All fine and well, but Java developers also need richer, simpler and more powerful abstractions that let them focus on the application they're trying to build, not the technology they're using. That's where Marge comes in. [Aside: Marge Project is named after Marge Simpson, yes **that** Marge Simpson. Why? Well - her hair is blue, big and tangled, something which the guys felt reflected the general state of Bluetooth for many Java developers. What a great name! :-)]

ANEXO C - Código fonte do Marge 0.4.0

C.1 net.java.dev.marge.communication.CommunicationChannel

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.communication;

import java.io.IOException;

import javax.microedition.io.Connection;

/**
 * Abstract representation of a communication channel. A communication
channel
 * uses a communication protocol that can be L2CAP or RFCOMM. A
communication
 * channel is used to manipulate the connection, receive and send messages.
 */
public abstract class CommunicationChannel {

	/**
	 * Closes the connection.
	 */
	public final void close() {
		this.disconnect();
	}

	/**
	 * Makes a disconnection.
	 */
```

```java
	protected abstract void disconnect();

	/**
	 * Receives a message.
	 * 
	 * @return byte[] Byte array representing the message received.
	 * @throws IOException
	 *             IOException is trown if a I/O problem happens during 
the
	 *             reading.
	 */
	public abstract byte[] receive() throws IOException;

	/**
	 * Sends a new message in a byte array format. It calls the 
sendMessage
	 * method.
	 * 
	 * @param message
	 *            The message that the device wants to send in a byte 
array
	 *            format.
	 * @throws IOException
	 *             IOException is trown if a I/O problem happens during 
the
	 *             sending.
	 */
	public void send(byte[] message) throws IOException {
		this.sendMessage(message);
	}

	/**
	 * Sends a message.
	 * 
	 * @param message
	 *            The message that the device wants to send in a byte 
array
	 *            format.
	 * @throws IOException
	 *             IOException is trown if a I/O problem happens during 
the
	 *             sending.
	 */
	protected abstract void sendMessage(byte[] message) throws 
IOException;

	/**
	 * Returns the current connection.
	 * 
	 * @return Current connection
	 */
	public abstract Connection getConnection();

}
```

C.2 net.java.dev.marge.communication.CommunicationListener

```java
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.communication;

import java.io.IOException;

/**
 * Listener interface that contains the methods that can be called during
the
 * communication process.
 */
public interface CommunicationListener {

	/**
	 * This method is called by the listener when the current device
receives a
	 * message sent by another.
	 *
	 * @param message
	 *            Byte array received representing the message.
	 */
	public abstract void receiveMessage(byte[] message);

	/**
	 * This method is called by the listener when the current device
tries to
	 * receive a message sent by another and a problem happens during
this
	 * process.
	 *
	 * @param e
	 *            IOException that can occur.
	 */
	public abstract void errorOnReceiving(IOException e);

	/**
	 * This method is called by the listener when the current device
tries to
	 * send a message to another and a problem happens during this
process.
```

```
	 *
	 * @param e
	 *            IOException that can occur.
	 */
	public abstract void errorOnSending(IOException e);

}
```

C.3 net.java.dev.marge.communication.ConnectionListener

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.communication;

import java.io.IOException;

import javax.bluetooth.RemoteDevice;

import net.java.dev.marge.entity.ServerDevice;

/**
 * Listener interface that contains the methods that can be called during
the
 * connection process.
 */
public interface ConnectionListener {

	/**
	 * This method is called by the listener when the current server
device
	 * receives a new connection represented by a remote device.
	 *
	 * @param serverDevice
	 *            The current <code>ServerDevice</code>.
	 * @param remoteDevice
```

```
	 *            The remote device that has connected on the server 
device.
	 */
	public void connectionEstablished(ServerDevice serverDevice,
			RemoteDevice remoteDevice);

	/**
	 * This method is called by the listener when the current server 
device
	 * tries to receive a new connection and problem happens during this
	 * process.
	 * 
	 * @param e
	 *            IOException that happened.
	 */
	public void errorOnConnection(IOException e);
}
```

C.4 net.java.dev.marge.communication.L2CAPCommunicationChannel

```
/*

 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 * 
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 * 
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 * 
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 * 
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.communication;

import java.io.IOException;

import javax.bluetooth.L2CAPConnection;
import javax.microedition.io.Connection;

/**
 * Representation of a L2CAP communication channel.
 */
public class L2CAPCommunicationChannel extends CommunicationChannel {
```

```java
	protected L2CAPConnection connection;

	protected boolean listening;

	/**
	 * Contructor that receives the L2CAP connection.
	 *
	 * @param connection
	 *             The L2CAP connection.
	 */
	public L2CAPCommunicationChannel(L2CAPConnection connection) {
		this.connection = connection;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see
net.java.dev.marge.communication.CommunicationChannel#receive()
	 */
	public byte[] receive() throws IOException {
		byte[] received = null;
		if (this.connection.ready()) {
			byte buffer[] = new
byte[this.connection.getReceiveMTU()];
			int bytesReceived = this.connection.receive(buffer);
			if (bytesReceived == 0) {
				return null;
			}
			received = new byte[bytesReceived];
			for (int i = 0; i < bytesReceived; i++) {
				received[i] = buffer[i];
			}
		}
		return received;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see
net.java.dev.marge.communication.CommunicationChannel#sendMessage(byte[])
	 */
	public void sendMessage(byte[] message) throws IOException {
		MessageSender sender = new MessageSender(message);
		sender.start();
		if (sender.getException() != null) {
			throw sender.getException();
		}
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see
net.java.dev.marge.communication.CommunicationChannel#disconnect()
	 */
	public void disconnect() {
		try {
			if (connection != null) {
				connection.close();
```

```
				}
			} catch (IOException e) {
			}
			connection = null;
		}

		/*
		 * (non-Javadoc)
		 *
		 * @see
net.java.dev.marge.communication.CommunicationChannel#getConnection()
		 */
		public Connection getConnection() {
			return connection;
		}

		/**
		 * Thread class that sends the messages.
		 */
		public final class MessageSender extends Thread {

			private byte[] message;

			private IOException ex;

			/**
			 * Contructor that receives the byte array message that will be
sent.
			 *
			 * @param message
			 *            Byte array to send.
			 */
			public MessageSender(byte[] message) {
				this.message = message;
				this.ex = null;
			}

			/*
			 * (non-Javadoc)
			 *
			 * @see java.lang.Thread#run()
			 */
			public void run() {
				try {
					connection.send(message);
				} catch (IOException e) {
					this.ex = e;
				}
			}

			/**
			 * Returns the IOException. This IOException is saved if a I/O
problem

			 * happens during it is triying to send the message and you can
get it

			 * by this method.
			 *
			 * @return IOExcepion instance.
			 */
			public IOException getException() {
				return ex;
```

```
		}
	}
}
```

C.5 net.java.dev.marge.communication.RFCOMMCommunicationChannel

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.communication;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import javax.microedition.io.Connection;
import javax.microedition.io.StreamConnection;

/**
 * Representation of a RFCOMM communication channel.
 */
public class RFCOMMCommunicationChannel extends CommunicationChannel {

	protected StreamConnection connection;

	protected DataOutputStream out;

	protected DataInputStream in;

	/**
	 * Contructor that receives the stream connection.
	 *
	 * @param connection
	 *            The stream connection.
	 * @throws IOException
	 *             IOException can occurs during it is trying to open the
input
```

```
	 *		and output.
	 */
	public RFCOMMCommunicationChannel(StreamConnection connection)
			throws IOException {
		this.connection = connection;
		this.in = connection.openDataInputStream();
		this.out = connection.openDataOutputStream();
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see
net.java.dev.marge.communication.CommunicationChannel#receive()
	 */
	public byte[] receive() throws IOException {
		int available = this.in.available();
		if (available == 0) {
			return null;
		}
		byte[] received = new byte[available];
		this.in.read(received);
		return received;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see
net.java.dev.marge.communication.CommunicationChannel#sendMessage(byte[])
	 */
	public void sendMessage(final byte[] message) throws IOException {
		MessageSender sender = new MessageSender(message);
		sender.start();
		if (sender.getException() != null) {
			throw sender.getException();
		}
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see
net.java.dev.marge.communication.CommunicationChannel#disconnect()
	 */
	public void disconnect() {
		try {
			if (in != null) {
				in.close();
			}
		} catch (IOException e) {
		}
		try {
			if (out != null) {
				out.close();
			}
		} catch (IOException e) {
		}
		try {
			if (connection != null) {
				connection.close();
```

```
				}
			} catch (IOException e) {
			}
			in = null;
			out = null;
			connection = null;
		}

		/*
		 * (non-Javadoc)
		 *
		 * @see
net.java.dev.marge.communication.CommunicationChannel#getConnection()
		 */
		public Connection getConnection() {
			return connection;
		}

		/**
		 * Return the data input stream of the connection used in the
constructor.
		 *
		 * @return DataInputStream The data input stream.
		 */
		public DataInputStream getDataInputStream() {
			return in;
		}

		/**
		 * Return the data output stream of the connection used in the
constructor.
		 *
		 * @return DataOutputStream The data input stream.
		 */
		public DataOutputStream getDataOutputStream() {
			return out;
		}

		/**
		 * Thread class that sends the messages.
		 */
		public final class MessageSender extends Thread {

			private byte[] message;

			private IOException ex;

			/**
			 * Contructor that receives the byte array message that will be
sent.
			 *
			 * @param message
			 *            Byte array to send.
			 */
			public MessageSender(byte[] message) {
				this.message = message;
				this.ex = null;
			}

			/*
			 * (non-Javadoc)
```

```java
             *
             * @see java.lang.Thread#run()
             */
            public void run() {
                  try {
                        out.write(this.message);
                        out.flush();
                  } catch (IOException e) {
                        this.ex = e;
                  }
            }

            /**
             * Returns the IOException. This IOException is saved if a I/O
problem
             * happens during it is triying to send the message and you can
get it
             * by this method.
             *
             * @return IOException instance.
             */
            public IOException getException() {
                  return ex;
            }
      }
}
```

C.6 net.java.dev.marge.entity.ClientDevice

```java
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.entity;

import net.java.dev.marge.communication.CommunicationChannel;
import net.java.dev.marge.communication.CommunicationListener;
```

```java
/**
 * Representation of a Bluetooth client device.
 */
public class ClientDevice extends Device {

	/**
	 * Contructor used to create a client. The incomming read messages of
the given Channel will be
	 * forwarded to the Listener at every Read Interval.
	 *
	 * @param communicationListener
	 *            <code>CommunicationListener</code> which will the
notified
	 *            for incoming messages.
	 * @param channel
	 *            <code>CommunicationChannel</code> used for
communication.
	 * @param readInterval
	 *            Time between each message reading.
	 */
	public ClientDevice(CommunicationListener communicationListener,
			CommunicationChannel channel, int readInterval) {
		super(communicationListener, channel, readInterval);
	}

}
```

C.7 net.java.dev.marge.entity.Device

```java
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.entity;
```

```
import java.io.IOException;
import java.util.Vector;

import net.java.dev.marge.communication.CommunicationChannel;
import net.java.dev.marge.communication.CommunicationListener;

/**
 * Abstract representation of a generic Bluetooth device.
 */
public abstract class Device implements Runnable {

	protected boolean enableBroadcast;

	protected Vector channels;

	private boolean listening;

	private CommunicationListener listener;

	private long readInterval;

	/**
	 * Constructor. The incomming read messages of the given Channel will
be
	 * forwarded to the Listener at every Read Interval.
	 *
	 * @param communicationListener
	 *             <code>CommunicationListener</code> which will the
notified
	 *             for incoming messages.
	 * @param channel
	 *             <code>CommunicationChannel</code> used for
communication.
	 * @param readInterval
	 *             Time between each message reading.
	 */
	public Device(CommunicationListener communicationListener,
			CommunicationChannel channel, int readInterval) {
		this.enableBroadcast = false;
		this.channels = new Vector(7);
		this.channels.addElement(channel);
		this.listener = communicationListener;
		this.listening = false;
		this.readInterval = readInterval;
	}

	/**
	 * Gets the CommunicationChannel at position i.
	 *
	 * @param i
	 *             Position of the desired Channel.
	 * @return <code>CommunicationChannel</code> at referenced position.
	 */
	protected CommunicationChannel getChannel(int i) {
		return (CommunicationChannel) this.channels.elementAt(i);
	}

	/**
	 * Sends the given bytes through the available Channels.
	 *
```

```java
 * @param message
 *            The bytes to be sent.
 */
public void send(byte[] message) {
      for (int i = 0; i < channels.size(); i++) {
            try {
                  this.getChannel(i).send(message);
            } catch (IOException e) {
                  this.listener.errorOnSending(e);
                  this.channels.removeElement(this.getChannel(i));
            }
      }
}

/**
 * Sends the given bytes through the available Channels, skiping the
 * specified one.
 *
 * @param message
 *            The bytes to be sent.
 * @param skip
 *            The Channel that will not receive the message.
 */
protected void send(byte[] message, CommunicationChannel skip) {
      for (int i = 0; i < channels.size(); i++) {
            if (!skip.equals(this.getChannel(i))) {
                  try {
                        this.getChannel(i).send(message);
                  } catch (IOException e) {
                        this.listener.errorOnSending(e);

this.channels.removeElement(this.getChannel(i));
                  }
            }
      }
}

/**
 * Starts listening for incoming messages.
 */
// XXX Two different Threads can't be opened.
public void startListening() {
      Thread t = new Thread(this);
      t.start();
}

/**
 * The run method that will start the listening process.
 */
// TODO Move to a inner class
public final void run() {
      if (!listening) {
            this.listening = true;
            this.listen();
      }
}

/**
 * Stops listening for incoming messages.
 */
public final void stopListenning() {
```

```java
			this.listening = false;
		}

		/**
		 * Listening for incoming messages.
		 */
		private final void listen() {
			while (listening) {
				for (int i = 0; i < this.channels.size(); i++) {
					try {
						byte[] b = this.getChannel(i).receive();
						if (b != null) {
							if (this.enableBroadcast) {
								this.send(b, this.getChannel(i));
							}
							this.listener.receiveMessage(b);
						}
					} catch (IOException e) {
						this.listener.errorOnReceiving(e);

		this.channels.removeElement(this.getChannel(i));
					}
				}
				try {
					Thread.sleep(this.readInterval);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}

		}

		/**
		 * Closes the device's communication channels.
		 */
		public void close() {
			for (int i = 0; i < this.channels.size(); i++) {
				this.getChannel(i).close();
			}
		}

		/**
		 * Returns the current read interval between every incoming message
	check.
		 *
		 * @return Read interval.
		 */
		public long getReadInterval() {
			return readInterval;
		}

		/**
		 * Sets the current read interval between every incoming message
	check.
		 */
		public void setReadInterval(long readInterval) {
			this.readInterval = readInterval;
		}

	}
```

C.8 net.java.dev.marge.entity.ServerDevice

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.entity;

import net.java.dev.marge.communication.CommunicationChannel;
import net.java.dev.marge.communication.CommunicationListener;

/**
 * Representation of a Bluetooth server device.
 */
public class ServerDevice extends Device {

	/**
	 * Contructor used to create a server. The incomming read messages of
the given Channel will be
	 * forwarded to the Listener at every Read Interval.
	 *
	 * @param communicationListener
	 *            <code>CommunicationListener</code> which will the
notified
	 *            for incoming messages.
	 * @param channel
	 *            <code>CommunicationChannel</code> used for
communication.
	 * @param readInterval
	 *            Time between each message reading.
	 */
	public ServerDevice(CommunicationListener communicationListener,
			CommunicationChannel channel, int readInterval) {
		super(communicationListener, channel, readInterval);
	}

	/**
	 * Enables message broadcast between connected devices.
	 *
```

```
         * @param enableBroadcast
         *              Parameter for enabling broadcast or not.
         */
        public void setEnableBroadcast(boolean enableBroadcast) {
                this.enableBroadcast = enableBroadcast;
        }

        /**
         * Adds a new communication channel into the server. It is used if
you want
         * to connect a new device.
         *
         * @param channel
         *              <code>CommunicationChannel</code> that will be added.
         */
        public void addChannel(CommunicationChannel channel) {
                // TODO check if not bigger than possible connections
                this.channels.addElement(channel);
        }
}
```

C.9 net.java.dev.marge.entity.config.ClientConfiguration

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.entity.config;

import javax.bluetooth.ServiceRecord;

import net.java.dev.marge.communication.CommunicationListener;

/**
 * Configuration for creating a <code>ClientDevice</code>. Hold all the
 * information used for creating a Client using the
 * <code>CommunicationFactory</code>.
 *
```

```
 */
public class ClientConfiguration extends Configuration {

	private ServiceRecord service;

	/**
	 * Constructor.
	 * 
	 * @param service
	 *            Remote Service that the client will connect.
	 * @param communicationListener
	 *            <code>CommunicationListener</code> which will the
notified
	 *            for incoming messages.
	 */
	public ClientConfiguration(ServiceRecord service,
			CommunicationListener communicationListener) {
		super(communicationListener);
		this.service = service;
	}

	/**
	 * Returns a URL to connect in the specified ServiceRecord.
	 * 
	 * @return Service connection URL.
	 */
	public String getConnectionURL() {
		// TODO see about encypt and authenticate
		return this.service.getConnectionURL(
				ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);
	}

}
```

C.10 net.java.dev.marge.entity.config.Configuration

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 * 
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 * 
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 * 
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 * 
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
```

```java
 */

package net.java.dev.marge.entity.config;

import net.java.dev.marge.communication.CommunicationListener;

/**
 * Base Configuration class for creating a new <code>Device</code>. This
 * class holds all basic information for creating a Device using the
 * <code>CommunicationFactory</code>.
 */
public abstract class Configuration {

	protected CommunicationListener communicationListener;

	protected int readInterval;

	/**
	 * Default constructor.
	 *
	 * @param communicationListener
	 *            Listener which will the notified for incoming messages.
	 */
	public Configuration(CommunicationListener communicationListener) {
		super();
		this.communicationListener = communicationListener;
		this.readInterval = 100;
	}

	/**
	 * Returns a Connection URL for use in the Generic Connection
Framework,
	 * following the current configurations.
	 *
	 * @return GCF connection URL.
	 */
	public abstract String getConnectionURL();

	/**
	 * Returns the current Listener that is notified on every new
incoming
	 * message.
	 *
	 * @return Listener for incomming messages.
	 */
	public CommunicationListener getCommunicationListener() {
		return communicationListener;
	}

	/**
	 * Sets the communication listener that will be notified on every new
	 * incoming message.
	 *
	 * @param communicationListener
	 *            <code>CommunicationListener</code> for messages.
	 */
	public void setCommunicationListener(
			CommunicationListener communicationListener) {
		this.communicationListener = communicationListener;
	}
```

```java
	/**
	 * Returns the current read interval between every incoming message check.
	 *
	 * @return Read interval.
	 */
	public int getReadInterval() {
		return readInterval;
	}

	/**
	 * Sets the current read interval between every incoming message check.
	 */
	public void setReadInterval(int readInterval) {
		this.readInterval = readInterval;
	}

}
```

C.11 net.java.dev.marge.entity.config.MargeDefaults

```java
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.entity.config;

// XXX Temporary before the UUID and Server Name generator.
/**
 * Interface with some default values used by the framework, if you do not
 * specify things in <code>Configuration</code>. It has the deafult UUID,
 * server name and service.
 */
```

```
public interface MargeDefaults {

	public static final String DEFAULT_UUID =
"102030405060708090A0B0C0D0E0F010";

	public static final String DEFAULT_SERVER_NAME =
"BTMargeDefaultServer";

	public static final int DEFAULT_SERVICE = 0x400000;
}
```

C.12 net.java.dev.marge.entity.config.ServerConfiguration

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.entity.config;

import javax.bluetooth.UUID;

import net.java.dev.marge.communication.CommunicationListener;

// TODO put attributes in a hashtable. The attributes would be setted by a
put
// method like available in the hashtable and by setEncript(),
setAuthorize, ...
/**
 * Configuration for creating a <code>ServerDevice</code>. Hold all the
 * information used for creating a Server using the
 * <code>CommunicationFactory</code>.
 *
 */
public class ServerConfiguration extends Configuration {

	private UUID uuid;
```

```
	private int service;

	private int maxNumberOfConnections;

	private String serverName;

	private String protocol;

	private boolean encrypt;

	private boolean authorize;

	private boolean authenticate;

	private String[] additionalParamsKeyValueToConnection;

	/**
	 * Constructor.
	 * 
	 * @param communicationListener
	 *            <code>CommunicationListener</code> which will the
notified for incoming messages.
	 */
	public ServerConfiguration(CommunicationListener
communicationListener) {
		super(communicationListener);
		// TODO create class that can generate service value, server
name and
		// UUID. Maybe using the MIDlet name and package as parameters
using
		// some hash alghoritm
		this.uuid = new UUID(MargeDefaults.DEFAULT_UUID, false);
		this.service = MargeDefaults.DEFAULT_SERVICE;
		this.serverName = MargeDefaults.DEFAULT_SERVER_NAME;
		this.encrypt = false;
		this.authorize = false;
		this.authenticate = false;
		this.maxNumberOfConnections = 1;
		this.protocol = "btssp";
	}

	/**
	 * Returns a URL used to create a Server Service.
	 * 
	 * @return Service creation URL.
	 */
	public String getConnectionURL() {
		String url = this.getProtocol() + "://localhost:"
				+ this.getUuid().toString() + ";name=" +
this.getServerName()
				+ ";encrypt=" + this.getEncrypt() + ";authorize="
				+ this.getAuthorize() + ";authenticate="
				+ this.getAuthenticate();

		String[] additionalParamsKeyValueToConnection = this
				.getAdditionalParamsKeyValueToConnection();
		if (additionalParamsKeyValueToConnection != null) {
			for (int i = 0; i <
additionalParamsKeyValueToConnection.length; i++) {
				url += ";" +
```

```java
additionalParamsKeyValueToConnection[i];
			}
		}
		return url;
	}

	/**
	 * Returns the Server name.
	 *
	 * @return Server name.
	 */
	public String getServerName() {
		return serverName;
	}

	/**
	 * Sets the name that will be used to create the server.
	 *
	 * @param serverName
	 *            New server name.
	 */
	public void setServerName(String serverName) {
		this.serverName = serverName;
	}

	/**
	 * Returns the number that will identify the Server service.
	 *
	 * @return Service identifier.
	 */
	public int getService() {
		return service;
	}

	/**
	 * Sets the number that will identify the Server service.
	 *
	 * @param service
	 *            Service identifier.
	 */
	public void setService(int service) {
		this.service = service;
	}

	/**
	 * Returns the current Universally Unique Identifier of the server.
	 *
	 * @return Universally Unique Identifier.
	 */
	public UUID getUuid() {
		return uuid;
	}

	/**
	 * Sets the server Universally Unique Identifier of the server.
	 *
	 * @param uuid
	 *            New Universally Unique Identifier.
	 */
	public void setUuid(UUID uuid) {
		this.uuid = uuid;
```

```
	}

	/**
	 * Returns the Bluetooth Stack protocol name that will be used to
create
	 * this Server.
	 *
	 * @return Bluetooth Stack protocol name.
	 */
	public String getProtocol() {
		return protocol;
	}

	/**
	 * Sets the Bluetooth Stack protocol name that will be used to create
this
	 * Server
	 *
	 * @param protocol
	 *            Bluetooth Stack protocol name.
	 */
	public void setProtocol(String protocol) {
		this.protocol = protocol;
	}

	/**
	 * Returns if the server will use authentication or not.
	 *
	 * @return flag.
	 */
	public boolean getAuthenticate() {
		return authenticate;
	}

	/**
	 * Sets if the server will use authentication.
	 *
	 * @param authenticate
	 */
	public void setAuthenticate(boolean authenticate) {
		this.authenticate = authenticate;
	}

	/**
	 * Returns the flag indicating if the server will need Authorization.
	 *
	 * @return Authorization flag.
	 */
	public boolean getAuthorize() {
		return authorize;
	}

	/**
	 * Sets the flag indicating if the server will need Authorization.
	 *
	 * @param authorize
	 *            Authorization flag.
	 */
	public void setAuthorize(boolean authorize) {
		this.authorize = authorize;
	}
```

```java
	/**
	 * Returns the flag indicating if the server will use data
Encryption.
	 *
	 * @return Encryption flag.
	 */
	public boolean getEncrypt() {
		return encrypt;
	}

	/**
	 * Sets the flag indicating if the server will use data Encryption.
	 *
	 * @param encrypt
	 *            Encryption flag.
	 */
	public void setEncrypt(boolean encrypt) {
		this.encrypt = encrypt;
	}

	/**
	 * Return any additional parameters that will be used in the server.
It does
	 * not inclued the Encrypt, Authorize and Authenticate parameters.
	 *
	 * @return Additional parameters.
	 */
	public String[] getAdditionalParamsKeyValueToConnection() {
		return additionalParamsKeyValueToConnection;
	}

	// XXX Like, new String[]{"receiveMTU=512", transmitMTU=512",
	// "master=true"};
	/**
	 * Sets the additional parameters that will be used in the server.
For
	 * Encrypt, Authorize and Authenticate parameters use the default
get's and
	 * set's.
	 *
	 * @param additionalParamsKeyValueToConnection
	 *            Additional parameters.
	 */
	public void setAdditionalParamsKeyValueToConnection(
			String[] additionalParamsKeyValueToConnection) {
		this.additionalParamsKeyValueToConnection =
additionalParamsKeyValueToConnection;
	}

	/**
	 * Returns the maximum number of connections that the Server will
support.
	 *
	 * @return Maximum number of connections.
	 */
	public int getMaxNumberOfConnections() {
		return maxNumberOfConnections;
	}

	/**
```

```
	 * Sets the maximum number of connections that the Server will
support. This
	 * number is limited by 7 by the Bluetooth technology.
	 *
	 * @param maxNumberOfConnections
	 *            The new maximum number of connections.
	 */
	public void setMaxNumberOfConnections(int maxNumberOfConnections) {
		this.maxNumberOfConnections = maxNumberOfConnections;
	}

}
```

C.13 net.java.dev.marge.factory.CommunicationFactory

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.factory;

import java.io.IOException;

import net.java.dev.marge.communication.ConnectionListener;
import net.java.dev.marge.entity.ClientDevice;
import net.java.dev.marge.entity.config.ClientConfiguration;
import net.java.dev.marge.entity.config.ServerConfiguration;

/**
 * Base methods for other Communication Factories create
 * <code>ClientDevice</code> and <code>ServerDevice</code>.
 */
public interface CommunicationFactory {
```

```java
	/**
	 * Creates a <code>ClientDevice</code> under the given
configurations.
	 *
	 * @param configuration
	 *            <code>ClientConfiguration</code> used to create the
client.
	 * @return Created <code>ClientDevice</code>.
	 * @throws IOException
	 *             if an I/O error occurs.
	 */
	public ClientDevice connectToServer(ClientConfiguration
configuration)
			throws IOException;

	/**
	 * Creates a <code>ServerDevice</code> with the given configurations.
The
	 * ServerDevice instance will be given to the
<code>ConnetionListener</code>
	 * when the connection is estabilished.
	 *
	 * @param configuration
	 *            <code>ServerConfiguration</code> used to create the
server.
	 * @param connectionListener
	 *            Listener that will be notified when a connection is
	 *            estabilished.
	 */
	public void waitClients(final ServerConfiguration configuration,
			final ConnectionListener connectionListener);
}
```

C.14 net.java.dev.marge.factory.L2CAPCommunicationFactory

```java
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
```

```java
 * http://marge.dev.java.net
 */

package net.java.dev.marge.factory;

import java.io.IOException;

import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.L2CAPConnection;
import javax.bluetooth.L2CAPConnectionNotifier;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.microedition.io.Connector;

import net.java.dev.marge.communication.CommunicationChannel;
import net.java.dev.marge.communication.ConnectionListener;
import net.java.dev.marge.communication.L2CAPCommunicationChannel;
import net.java.dev.marge.entity.ClientDevice;
import net.java.dev.marge.entity.ServerDevice;
import net.java.dev.marge.entity.config.ClientConfiguration;
import net.java.dev.marge.entity.config.ServerConfiguration;

/**
 * Create Client and Server Devices using the Bluetooth L2CAP Protocol.
 */
public class L2CAPCommunicationFactory implements CommunicationFactory {

      private static final String PROTOCOL_NAME = "btl2cap";

      /**
       * Creates a <code>ClientDevice</code> under the given
configurations. The
       * Client will connect to the server using the L2CAP Protocol.
       *
       * @param configuration
       *             <code>ClientConfiguration</code> used to create the
client.
       * @return Created <code>ClientDevice</code>.
       * @throws IOException
       *              if an I/O error occurs.
       */
      public ClientDevice connectToServer(ClientConfiguration
configuration)
                  throws IOException {

            ClientDeviceFactory clientFactory = new ClientDeviceFactory(
                        configuration);
            clientFactory.start();

            synchronized (this) {
                  try {
                        this.wait();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }

            if (clientFactory.getException() != null) {
                  throw clientFactory.getException();
            }
```

```
		return clientFactory.getDevice();
	}

	/**
	 * Creates a <code>ServerDevice</code> with the given configurations.
The
	 * ServerDevice instance will be given to the
<code>ConnetionListener</code>
	 * when the connection is estabilished and it will use the L2CAP
Protocol.
	 * 
	 * @param configuration
	 *            <code>ServerConfiguration</code> used to create the
server.
	 * @param connectionListener
	 *            Listener that will be notified when a connection is
	 *            estabilished.
	 */
	public void waitClients(final ServerConfiguration configuration,
			final ConnectionListener listener) {
		new Thread() {
			public void run() {
				try {
					LocalDevice.getLocalDevice().setDiscoverable(
							DiscoveryAgent.GIAC);
					configuration.setProtocol(PROTOCOL_NAME);

					L2CAPConnectionNotifier conn =
(L2CAPConnectionNotifier) Connector

	.open(configuration.getConnectionURL());
					L2CAPConnection connection =
conn.acceptAndOpen();
					L2CAPCommunicationChannel channel = new
L2CAPCommunicationChannel(
							connection);

					RemoteDevice remoteDevice = RemoteDevice

	.getRemoteDevice(channel.getConnection());

					ServerDevice serverDevice = new
ServerDevice(configuration
							.getCommunicationListener(),
channel, configuration
							.getReadInterval());

					listener.connectionEstablished(serverDevice,
remoteDevice);

					int connections = 0;
					while (++connections < configuration
							.getMaxNumberOfConnections()) {

						connection = conn.acceptAndOpen();
						channel = new
L2CAPCommunicationChannel(connection);
						remoteDevice =
RemoteDevice.getRemoteDevice(channel
								.getConnection());
```

```
                                        serverDevice.addChannel(channel);


        listener.connectionEstablished(serverDevice,
                                                        remoteDevice);
                                }

                        } catch (IOException e) {
                                e.printStackTrace();
                                listener.errorOnConnection(e);
                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                }
            }.start();
        }

        final public class ClientDeviceFactory extends Thread {

                private IOException exception;

                private ClientConfiguration configuration;

                private ClientDevice device;

                public ClientDeviceFactory(ClientConfiguration configuration) {
                        this.configuration = configuration;
                        this.exception = null;
                }

                public void run() {
                        try {
                                CommunicationChannel channel = new
L2CAPCommunicationChannel(
                                        (L2CAPConnection)
Connector.open(configuration
                                                        .getConnectionURL()));

                                device = new ClientDevice(configuration
                                        .getCommunicationListener(), channel,
configuration
                                        .getReadInterval());
                        } catch (IOException e) {
                                exception = e;
                        }
                        synchronized (L2CAPCommunicationFactory.this) {
                                L2CAPCommunicationFactory.this.notify();
                        }
                }

                public ClientDevice getDevice() {
                        return device;
                }

                public IOException getException() {
                        return exception;
                }
        }
}
```

C.15 net.java.dev.marge.factory.RFCOMMCommunicationFactory

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.factory;

import java.io.IOException;

import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.microedition.io.Connector;
import javax.microedition.io.StreamConnection;
import javax.microedition.io.StreamConnectionNotifier;

import net.java.dev.marge.communication.CommunicationChannel;
import net.java.dev.marge.communication.ConnectionListener;
import net.java.dev.marge.communication.RFCOMMCommunicationChannel;
import net.java.dev.marge.entity.ClientDevice;
import net.java.dev.marge.entity.ServerDevice;
import net.java.dev.marge.entity.config.ClientConfiguration;
import net.java.dev.marge.entity.config.ServerConfiguration;

/**
 * Create Client and Server Devices using the Bluetooth RFCOMM Protocol.
 */
public class RFCOMMCommunicationFactory implements CommunicationFactory {

      private static final String PROTOCOL_NAME = "btspp";

      /**
       * Creates a <code>ClientDevice</code> under the given
configurations. The
       * Client will connect to the server using the RFCOMM Protocol.
```

```
	 *
	 * @param configuration
	 *            <code>ClientConfiguration</code> used to create the
client.
	 * @return Created <code>ClientDevice</code>.
	 * @throws IOException
	 *             if an I/O error occurs.
	 */
	public ClientDevice connectToServer(ClientConfiguration
configuration)
			throws IOException {

		ClientDeviceFactory clientFactory = new ClientDeviceFactory(
				configuration);
		clientFactory.start();

		synchronized (this) {
			try {
				this.wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}

		if (clientFactory.getException() != null) {
			throw clientFactory.getException();
		}

		return clientFactory.getDevice();
	}

	/**
	 * Creates a <code>ServerDevice</code> with the given configurations.
The
	 * ServerDevice instance will be given to the
<code>ConnetionListener</code>
	 * when the connection is estabilished and it will use the RFCOMM
Protocol.
	 *
	 * @param configuration
	 *            <code>ServerConfiguration</code> used to create the
server.
	 * @param connectionListener
	 *            Listener that will be notified when a connection is
	 *            estabilished.
	 */
	public void waitClients(final ServerConfiguration configuration,
			final ConnectionListener listener) {
		new Thread() {
			public void run() {
				try {
					LocalDevice.getLocalDevice().setDiscoverable(
							DiscoveryAgent.GIAC);
					configuration.setProtocol(PROTOCOL_NAME);

					StreamConnectionNotifier conn =
(StreamConnectionNotifier) Connector

	.open(configuration.getConnectionURL());
					StreamConnection connection =
conn.acceptAndOpen();
```

```
                              RFCOMMCommunicationChannel channel = new
RFCOMMCommunicationChannel(
                                          connection);

                              RemoteDevice remoteDevice = RemoteDevice

      .getRemoteDevice(channel.getConnection());

                              ServerDevice serverDevice = new
ServerDevice(configuration
                                          .getCommunicationListener(),
channel, configuration
                                          .getReadInterval());

                              listener.connectionEstablished(serverDevice,
remoteDevice);

                              int connections = 0;
                              while (++connections < configuration
                                          .getMaxNumberOfConnections()) {

                                    connection = conn.acceptAndOpen();
                                    channel = new
RFCOMMCommunicationChannel(connection);
                                    remoteDevice =
RemoteDevice.getRemoteDevice(channel
                                                .getConnection());

                                    serverDevice.addChannel(channel);


      listener.connectionEstablished(serverDevice,
                                                remoteDevice);
                              }

                        } catch (IOException e) {
                              e.printStackTrace();
                              listener.errorOnConnection(e);
                        } catch (Exception e) {
                              e.printStackTrace();
                        }
                  }
            }.start();
      }

      final public class ClientDeviceFactory extends Thread {

            private IOException exception;

            private ClientConfiguration configuration;

            private ClientDevice device;

            public ClientDeviceFactory(ClientConfiguration configuration) {
                  this.configuration = configuration;
                  this.exception = null;
            }

            public void run() {
                  try {
                        CommunicationChannel channel = new
```

```
RFCOMMCommunicationChannel(
                                    (StreamConnection)
Connector.open(configuration
                                                .getConnectionURL()));

                        device = new ClientDevice(configuration
                                    .getCommunicationListener(), channel,
configuration
                                    .getReadInterval());
                  } catch (IOException e) {
                        exception = e;
                  }
                  synchronized (RFCOMMCommunicationFactory.this) {
                        RFCOMMCommunicationFactory.this.notify();
                  }
            }

            public ClientDevice getDevice() {
                  return device;
            }

            public IOException getException() {
                  return exception;
            }
      }
}
```

C.16 net.java.dev.marge.inquiry.DefaultDiscoveryListener

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.inquiry;
```

```java
import java.util.Vector;

import javax.bluetooth.DeviceClass;
import javax.bluetooth.DiscoveryListener;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.ServiceRecord;

/**
 * Marge's Service and Device Search Listener implementation. It iss used
in the
 * <code>DeviceDiscoverer</code> and <code>ServiceDiscoverer</code>
classes.
 */
public class DefaultDiscoveryListener implements DiscoveryListener {

	private boolean inquiryCompleted;

	private ServiceSearchListener sListener;

	private InquiryListener iListener;

	private Vector foundDevices;

	private ServiceRecord[] foundServices;

	private RemoteDevice remoteDevice;

	private DefaultDiscoveryListener() {
		this.inquiryCompleted = true;
	}

	/**
	 * Constructor for Service Search.
	 *
	 * @param remoteDevice
	 *            Device to be scanned.
	 * @param slistener
	 *            ServiceSearchListener to be notified of found Services.
	 */
	public DefaultDiscoveryListener(RemoteDevice remoteDevice,
			ServiceSearchListener slistener) {
		this();
		this.remoteDevice = remoteDevice;
		this.sListener = slistener;
	}

	/**
	 * Constructor for Device Search.
	 *
	 * @param iListener
	 *            Listener notified of found Devices.
	 */
	public DefaultDiscoveryListener(InquiryListener iListener) {
		this();
		this.foundDevices = new Vector(5);
		this.iListener = iListener;
	}

	/**
	 * Returns the status of the Device Discovery process, saying if it
has
```

```java
         * Finished.
         *
         * @return Status of Device Discovery.
         */
        public boolean hasInquiryFinished() {
              return this.inquiryCompleted;
        }

        /**
         * Returns the devices discovered in the Search Process.
         *
         * @return Devices discovered.
         */
        public RemoteDevice[] getDiscoveredDevices() {
              RemoteDevice devices[] = new
RemoteDevice[this.foundDevices.size()];
              this.foundDevices.copyInto(devices);
              return devices;
        }

        /**
         * Marge implementation of <code>DiscoveryListener</code>. Called
when a
         * device is found during an inquiry. An inquiry searches for devices
that
         * are discoverable. The same device may be returned multiple times.
         *
         * @param device
         *            The device that was found during the inquiry.
         * @param dclass
         *            The service classes, major device class, and minor
device
         *            class of the remote device.
         */
        public void deviceDiscovered(RemoteDevice device, DeviceClass dclass)
{
              if (!this.foundDevices.contains(device)) {
                    this.foundDevices.addElement(device);
              }
              this.iListener.deviceDiscovered(device, dclass);
        }

        /**
         * Marge implementation of <code>DiscoveryListener</code>. Called
when an
         * inquiry is completed. The discType will be INQUIRY_COMPLETED if
the
         * inquiry ended normally or INQUIRY_TERMINATED if the inquiry was
canceled
         * by a call to DiscoveryAgent.cancelInquiry(). The discType will be
         * INQUIRY_ERROR if an error occurred while processing the inquiry
causing
         * the inquiry to end abnormally.
         *
         * @param discType
         *            The type of request that was completed; either
         *            INQUIRY_COMPLETED, INQUIRY_TERMINATED, or
INQUIRY_ERROR.
         */
        public void inquiryCompleted(int discType) {
              this.inquiryCompleted = true;
```

```
		switch (discType) {
		case INQUIRY_COMPLETED:

	this.iListener.inquiryCompleted(this.getDiscoveredDevices());
			break;
		case INQUIRY_TERMINATED:
			break;
		case INQUIRY_ERROR:
			this.iListener.inquiryError();
			break;
		}
	}

	/**
	 * Marge implementation of <code>DiscoveryListener</code>. Called
when a
	 * service search is completed or was terminated because of an error.
	 *
	 * @param transID
	 *            The transaction ID identifying the request which
initiated the
	 *            service search.
	 * @param respCode
	 *            The response code that indicates the status of the
	 *            transaction.
	 */
	public void serviceSearchCompleted(int transID, int respCode) {
		switch (respCode) {
		case DiscoveryListener.SERVICE_SEARCH_COMPLETED:
			this.sListener.serviceSearchCompleted(this.remoteDevice,
					this.foundServices);
			break;
		case DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHABLE:
			this.sListener.deviceNotReachable();
			break;
		case DiscoveryListener.SERVICE_SEARCH_NO_RECORDS:
			this.sListener.serviceSearchCompleted(this.remoteDevice,
					new ServiceRecord[] {});
			break;
		case DiscoveryListener.SERVICE_SEARCH_TERMINATED:
			break;
		case DiscoveryListener.SERVICE_SEARCH_ERROR:
			this.sListener.serviceSearchError();
			break;
		}
	}

	/**
	 * Marge implementation of <code>DiscoveryListener</code>. Called
when
	 * service(s) are found during a service search.
	 *
	 * @param transID
	 *            The transaction ID of the service search that is
posting the
	 *            result.
	 * @param services
	 *            A list of services found during the search request.
	 */
	public void servicesDiscovered(int transID, ServiceRecord[] services)
{
```

```
		this.foundServices = services;
	}

}
```

C.17 net.java.dev.marge.inquiry.DeviceDiscoverer

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.inquiry;

import javax.bluetooth.BluetoothStateException;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;

/**
 * Provides methods to search Bluetooth enabled devices in the area.
 */
public class DeviceDiscoverer {

	private static DeviceDiscoverer instance;

	private DiscoveryAgent agent;

	private DefaultDiscoveryListener listener;

	/**
	 * Constructor. To get a instance of this class use the static
getInstance()
	 * method.
	 *
	 * @throws BluetoothStateException
	 *              if can't get the Local Device.
```

```java
 */
private DeviceDiscoverer() throws BluetoothStateException {
      this.agent = LocalDevice.getLocalDevice().getDiscoveryAgent();
      this.listener = null;
}

// TODO create method for each of the 3 types
/**
 * Starts a inquiry for devices.
 *
 * @param type
 *           The type of inquiry, use DiscoveryAgent constants.
 * @param inquiryListener
 *           The <code>InquiryListener</code> that will be notified.
 * @throws BluetoothStateException
 *            An exception can be trown during the inquiry.
 */
public void startInquiry(int type, InquiryListener inquiryListener)
            throws BluetoothStateException {
      this.listener = new DefaultDiscoveryListener(inquiryListener);
      this.agent.startInquiry(type, this.listener);
}

// TODO method for cached and preknown
/**
 * Retrieves devices previously known.
 *
 * @param type
 *           The type of retrieve, use DiscoveryAgent constants.
 * @return RemoteDevice An array of remote devices.
 */
public RemoteDevice[] retrieveDevices(int type) {
      return this.agent.retrieveDevices(type);
}

/**
 * Gets an array of RemoteDevice discovered.
 *
 * @return Array of RemoteDevice.
 */
public RemoteDevice[] getDiscoveredDevices() {
      RemoteDevice[] devices;
      if (this.listener == null) {
            devices = new RemoteDevice[0];
      } else {
            devices = this.listener.getDiscoveredDevices();
      }
      return devices;
}

/**
 * Cancels the inquiry.
 */
public void cancelInquiry() {
      if (this.listener != null) {
            this.agent.cancelInquiry(this.listener);
            this.listener = null;
      }
}

/**
```

```
     * Checks if the inquiry has finished.
     *
     * @return True if the inquiry has finished, false if not yet.
     */
    public boolean hasInquiryFinished() {
        boolean response;
        if (this.listener == null) {
            response = true;
        } else {
            response = this.listener.hasInquiryFinished();
        }
        return response;
    }

    /**
     * Returns a Singleton instance of DeviceDiscoverer.
     *
     * @return Singleton DeviceDiscoverer instance.
     */
    public static DeviceDiscoverer getInstance() throws
BluetoothStateException {
        if (instance == null) {
            instance = new DeviceDiscoverer();
        }
        return instance;
    }

}
```

C.18 net.java.dev.marge.inquiry.InquiryListener

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */
```

```
package net.java.dev.marge.inquiry;

import javax.bluetooth.DeviceClass;
import javax.bluetooth.RemoteDevice;

/**
 * Listener interface that contains the methods that can be called during
the
 * inquiry for devices process.
 */
public interface InquiryListener {

	/**
	 * This method is called by the listener when a device is discovered
during
	 * the inquiry process.
	 *
	 * @param device
	 *            The remote device that was discovered.
	 * @param deviceClass
	 *            The device class.
	 */
	public void deviceDiscovered(RemoteDevice device, DeviceClass
deviceClass);

	/**
	 * This method is called by the listener when the inquiry for devices
is
	 * completed.
	 *
	 * @param devices
	 *            An array of remote devices found during the inquiry
process.
	 */
	public void inquiryCompleted(RemoteDevice[] devices);

	/**
	 * This method is called by the listener when an error happens during
the
	 * inquiry for devices process.
	 */
	public void inquiryError();
}
```

C.19 net.java.dev.marge.inquiry.ServiceDiscoverer

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
```

```
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.inquiry;

import javax.bluetooth.BluetoothStateException;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.UUID;

/**
 * Provides methods to performs Service Discovery in Remote Devices.
 */
public class ServiceDiscoverer {

	// TODO make possible to get the response condition
	// (serviceSearchCompleted)

	private static ServiceDiscoverer instance;

	private DiscoveryAgent agent;

	private DefaultDiscoveryListener listener;

	private int transID;

	/**
	 * Constructor. To get a instance of this class use the static
getInstance
	 * method.
	 *
	 * @throws BluetoothStateException
	 *             if can't get the Local Device.
	 */
	private ServiceDiscoverer() throws BluetoothStateException {
		this.agent = LocalDevice.getLocalDevice().getDiscoveryAgent();
		this.listener = null;
	}

	/**
	 * Performs a search of the given UUID's in the especified
RemoteDevice. The
	 * search result will be notified to the
<code>ServiceSearchListener</code>.
	 *
	 * @param uuids
	 *            UUID's to be found.
	 * @param remoteDevice
```

```java
         *              Device to be searched.
         * @param listener
         *              Search result Listener.
         * @throws BluetoothStateException
         *               if can not use Bluetooth Search.
         */
        public void startSearch(UUID uuids[], RemoteDevice remoteDevice,
                        ServiceSearchListener listener) throws
BluetoothStateException {
                this.listener = new DefaultDiscoveryListener(remoteDevice,
listener);
                this.transID = this.agent.searchServices(null, uuids,
remoteDevice,
                                this.listener);
        }

        /**
         * Stops a search that has been started using startSearch.
         */
        public void cancelSearch() {
                if (this.listener != null) {
                        this.agent.cancelServiceSearch(this.transID);
                        this.listener = null;
                }
        }

        // TODO public ServiceRecord[] startSearch(RemoteDevice device)
        /**
         * Returns a Singleton instance of ServiceDiscoverer.
         *
         * @return Singleton ServiceDiscoverer instance.
         */
        public static ServiceDiscoverer getInstance()
                        throws BluetoothStateException {
                if (instance == null) {
                        instance = new ServiceDiscoverer();
                }
                return instance;
        }

}
```

C.20 net.java.dev.marge.inquiry.ServiceSearchListener

```java
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
```

```java
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.inquiry;

import javax.bluetooth.RemoteDevice;
import javax.bluetooth.ServiceRecord;

/**
 * Listener interface that contains the methods that can be called during
the
 * service search process.
 */
public interface ServiceSearchListener {

	/**
	 * This method is called by the listener when the search process has
been
	 * completed in a remote device.
	 *
	 * @param remoteDevice
	 *             The remote device that was searched.
	 * @param services
	 *             The services found.
	 */
	public void serviceSearchCompleted(RemoteDevice remoteDevice,
			ServiceRecord services[]);

	/**
	 * This method is called by the listener when a remote device could
not be
	 * reachable.
	 */
	public void deviceNotReachable();

	/**
	 * This method is called by the listener when an error happens during
the
	 * search process.
	 */
	public void serviceSearchError();

}
```

ANEXO D - Código fonte do Blue Chat usando o Marge 0.4.0

D.1 net.java.dev.marge.chat.ChatMIDlet

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.chat;

import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import net.java.dev.marge.chat.ui.MainMenu;
import net.java.dev.marge.entity.Device;

public class ChatMIDlet extends MIDlet {

	public static ChatMIDlet instance;

	private Display display;

	private Device device;

	private MainMenu mainMenu;

	public ChatMIDlet() {
		display = Display.getDisplay(this);
		instance = this;
	}

	protected void destroyApp(boolean arg0) throws
MIDletStateChangeException {
		notifyDestroyed();
```

```
	}

	protected void pauseApp() {
	}

	protected void startApp() throws MIDletStateChangeException {
		setCurrent(this.mainMenu = new MainMenu());
	}

	public void exit() {
		try {
			this.destroyApp(true);
		} catch (MIDletStateChangeException e) {
			e.printStackTrace();
		}
	}

	public void setCurrent(Displayable d) {
		this.display.setCurrent(d);
	}

	public Device getDevice() {
		return device;
	}

	public void setDevice(Device device) {
		this.device = device;
	}

	public void showError(String message, Displayable d) {
		Alert alert = new Alert("Erro", message, null,
AlertType.ERROR);
		alert.setTimeout(2000);
		display.setCurrent(alert, d);

	}

	public void showMainMenu() {
		this.setCurrent(this.mainMenu);
	}

}
```

D.2 net.java.dev.marge.chat.ui.AboutScreen

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */
package net.java.dev.marge.chat.ui;


import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;

import net.java.dev.marge.chat.ChatMIDlet;
import net.java.dev.marge.util.ImageUtil;

public class AboutScreen extends Form implements CommandListener {

	private Displayable previous;

	public AboutScreen(Displayable previous) {
		super("About");
		this.previous = previous;
		this.setCommandListener(this);
		this.addCommand(new Command("Back", Command.BACK, 1));
		this.append("Blue Chat");
		this
				.append("This demo is open source and part of
Project Marge.");
		this
				.append("Marge is a framework to help the
development of Bluetooth applications in Java");

		this.append(ImageUtil.loadImage("/marge.png"));
		this
				.append("For more information visit the website:
http://marge.dev.java.net");
		this.append(ImageUtil.loadImage("/mobileembedded.png"));
	}

	public void commandAction(Command arg0, Displayable arg1) {
		ChatMIDlet.instance.setCurrent(this.previous);
	}

}
```

D.3 net.java.dev.marge.chat.ui.ChatRoom

```
/*
```

```java
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.chat.ui;

import java.io.IOException;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.StringItem;
import javax.microedition.lcdui.TextField;

import net.java.dev.marge.chat.ChatMIDlet;
import net.java.dev.marge.communication.CommunicationListener;
import net.java.dev.marge.entity.Device;

public class ChatRoom extends Form implements CommunicationListener,
            CommandListener {

      private Device device;

      private Command back;

      private TextField chatField;

      public ChatRoom() {
            super("Chat...");
            this.chatField = new TextField("Message", null, 100,
TextField.ANY);
            this.append(this.chatField);
            this.addCommand(new Command("Send", Command.OK, 1));
            this.addCommand(this.back = new Command("Back", Command.BACK,
1));
            this.setCommandListener(this);
      }

      public void receiveMessage(byte[] receivedString) {
            this.insert(1, new StringItem("Received: ", new
String(receivedString)));
      }
```

```java
	public void errorOnReceiving(IOException e) {
		e.printStackTrace();
		this.leaveChat();
	}

	public void errorOnSending(IOException e) {
		e.printStackTrace();
		this.leaveChat();
	}

	public void sendMessage(String message) {
		this.device.send(message.getBytes());
		this.insert(1, new StringItem("Sent: ", message));
	}

	public void setDevice(Device device) {
		this.device = device;
	}

	public void leaveChat() {
		this.deleteAll();
		this.append(this.chatField);
		this.device.close();
		ChatMIDlet.instance.showMainMenu();
	}

	public void commandAction(Command c, Displayable d) {
		if (c == this.back) {
			this.leaveChat();
		} else {
				this.sendMessage(this.chatField.getString());
		}
	}
}
```

D.4 net.java.dev.marge.chat.ui.InquiryScreen

```java
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
```

```java
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.chat.ui;

import java.io.IOException;
import java.util.Vector;

import javax.bluetooth.BluetoothStateException;
import javax.bluetooth.DeviceClass;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.ServiceRecord;
import javax.bluetooth.UUID;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.List;

import net.java.dev.marge.chat.ChatMIDlet;
import net.java.dev.marge.entity.config.ClientConfiguration;
import net.java.dev.marge.entity.config.MargeDefaults;
import net.java.dev.marge.factory.CommunicationFactory;
import net.java.dev.marge.inquiry.DeviceDiscoverer;
import net.java.dev.marge.inquiry.InquiryListener;
import net.java.dev.marge.inquiry.ServiceDiscoverer;
import net.java.dev.marge.inquiry.ServiceSearchListener;

public class InquiryScreen extends List implements CommandListener,
            ServiceSearchListener, InquiryListener {

      private static final UUID SERVICE_UUID = new UUID(
                  MargeDefaults.DEFAULT_UUID, false);

      private MainMenu menu;

      private ServiceDiscoverer serviceDiscoverer;

      private DeviceDiscoverer deviceDiscoverer;

      private Command select;

      private Command stopOrBack;

      private Vector devices;

      private CommunicationFactory factory;

      public InquiryScreen(MainMenu menu, CommunicationFactory factory)
                  throws BluetoothStateException {
            super("Inquirying...", List.IMPLICIT);
            this.select = null;
            this.menu = menu;
            this.factory = factory;
            this.serviceDiscoverer = ServiceDiscoverer.getInstance();
```

```
            this.deviceDiscoverer = DeviceDiscoverer.getInstance();
            this.devices = new Vector(5);
            this.addCommand(this.stopOrBack = new Command("Stop",
                        Command.CANCEL, 1));
            this.setCommandListener(this);
            new Thread() {
                  public void run() {
                        try {

      InquiryScreen.this.deviceDiscoverer.startInquiry(
                                                DiscoveryAgent.GIAC,
InquiryScreen.this);
                        } catch (BluetoothStateException e) {
                              e.printStackTrace();
                        }
                  }
            }.start();
      }

      public void commandAction(Command c, Displayable arg1) {
            if (c == this.stopOrBack) {
                  if (c.getCommandType() == Command.CANCEL) {
                        this.deviceDiscoverer.cancelInquiry();
                        this.removeCommand(c);
                        this.stopOrBack = new Command("Back", Command.BACK,
1);

                        this.addCommand(this.stopOrBack);
                  } else {
                        ChatMIDlet.instance.showMainMenu();
                  }
            } else {
                  if (this.select != null) {
                        this.deviceDiscoverer.cancelInquiry();
                        try {
                              serviceDiscoverer.startSearch(new UUID[] {
SERVICE_UUID },
                                                (RemoteDevice)
this.devices.elementAt(this
                                                            .getSelectedIndex()),
this);
                        } catch (BluetoothStateException e) {
                              e.printStackTrace();
                        }
                  }
            }
      }

      public void deviceNotReachable() {
      }

      public void serviceSearchCompleted(RemoteDevice remoteDevice,
                  ServiceRecord[] services) {

            ClientConfiguration config = new
ClientConfiguration(services[0],
                        this.menu.getRoom());
            try {

this.menu.connectionEstablished(this.factory.connectToServer(config));
            } catch (IOException e) {
```

```
                        this.menu.errorOnConnection(e);
                }
        }

        public void serviceSearchError() {
        }

        public void deviceDiscovered(RemoteDevice device, DeviceClass
deviceClass) {
                if (this.select == null) {
                        this.select = new Command("Selec", Command.OK, 1);
                        this.addCommand(select);
                }

                this.devices.addElement(device);
                this.setTitle("Inquirying...  " + this.devices.size());
                try {
                        this.append(device.getFriendlyName(false), null);
                } catch (IOException e) {
                        this.append(device.getBluetoothAddress(), null);
                        e.printStackTrace();
                }
        }

        public void inquiryCompleted(RemoteDevice[] devices) {
                this.setTitle(Integer.toString(this.devices.size()) + " devices
found");
                this.removeCommand(this.stopOrBack);
                this.stopOrBack = new Command("Back", Command.BACK, 1);
                this.addCommand(this.stopOrBack);
        }

        public void inquiryError() {

        }

}
```

D.5 net.java.dev.marge.chat.ui.LoadingScreen

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
```

```
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.chat.ui;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Gauge;

import net.java.dev.marge.chat.ChatMIDlet;

public class LoadingScreen extends Form implements CommandListener {

	private MainMenu menu;

	public LoadingScreen(MainMenu menu) {
		super("Waiting...");
		this.menu = menu;
		this.append(new Gauge("Waiting for connections", false,
Gauge.INDEFINITE, Gauge.CONTINUOUS_RUNNING));
		this.addCommand(new Command("Cancel", Command.CANCEL, 1));
		this.setCommandListener(this);
	}

	public void commandAction(Command arg0, Displayable arg1) {
		this.menu.setCancelled();
		ChatMIDlet.instance.showMainMenu();
	}
}
```

D.6 net.java.dev.marge.chat.ui.MainMenu

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
```

```
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.chat.ui;

import java.io.IOException;

import javax.bluetooth.BluetoothStateException;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.List;

import net.java.dev.marge.chat.ChatMIDlet;
import net.java.dev.marge.communication.ConnectionListener;
import net.java.dev.marge.entity.ClientDevice;
import net.java.dev.marge.entity.ServerDevice;
import net.java.dev.marge.entity.config.ServerConfiguration;
import net.java.dev.marge.factory.CommunicationFactory;
import net.java.dev.marge.factory.L2CAPCommunicationFactory;
import net.java.dev.marge.factory.RFCOMMCommunicationFactory;

public class MainMenu extends List implements CommandListener,
            ConnectionListener {

      private final String SELEC_COMMAND_NAME = "Select";

      private ChatRoom room;

      private boolean cancelled;

      public MainMenu() {
            super("Blue Chat", List.IMPLICIT);

            this.append("Run RFCOMM client", null);
            this.append("Run L2CAP client", null);
            this.append("Run RFCOMM server", null);
            this.append("Run L2CAP server", null);
            this.append("About", null);
            this.append("Exit", null);

            this.addCommand(new Command(SELEC_COMMAND_NAME, Command.OK,
1));
            this.setCommandListener(this);
            this.room = new ChatRoom();
            this.cancelled = false;
      }

      public void commandAction(Command c, Displayable d) {
            try {
```

```
        LocalDevice.getLocalDevice().setDiscoverable(DiscoveryAgent.GIAC);
                switch (this.getSelectedIndex()) {
                case 0:
                        this.performInquiry(new
RFCOMMCommunicationFactory());
                        break;
                case 1:
                        this.performInquiry(new
L2CAPCommunicationFactory());
                        break;
                case 2:
                        this.cancelled = false;
                        ChatMIDlet.instance.setCurrent(new
LoadingScreen(this));
                        CommunicationFactory factory = new
RFCOMMCommunicationFactory();
                        ServerConfiguration config = new
ServerConfiguration(room);
                        config.setMaxNumberOfConnections(8);
                        factory.waitClients(config, this);
                        break;
                case 3:
                        this.cancelled = false;
                        ChatMIDlet.instance.setCurrent(new
LoadingScreen(this));
                        factory = new L2CAPCommunicationFactory();
                        config = new ServerConfiguration(room);
                        config.setMaxNumberOfConnections(8);
                        factory.waitClients(config, this);
                        break;
                case 4:
                        ChatMIDlet.instance.setCurrent(new
AboutScreen(this));
                        break;
                default:
                        ChatMIDlet.instance.exit();
                        break;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        public void performInquiry(CommunicationFactory factory) {
            try {
                ChatMIDlet.instance.setCurrent(new InquiryScreen(this,
factory));
            } catch (BluetoothStateException e) {
                e.printStackTrace();
            }
        }

        public ChatRoom getRoom() {
            return room;
        }

        public void errorOnConnection(IOException exception) {
            ChatMIDlet.instance.showError(exception.getMessage(), this);
        }

        public void connectionEstablished(ClientDevice device) {
```

```
			if (!this.cancelled) {
				device.startListening();
				this.room.setDevice(device);
				ChatMIDlet.instance.setCurrent(this.room);
			}
		}

		public void connectionEstablished(ServerDevice device, RemoteDevice
remote) {
			System.out.println(remote.getBluetoothAddress());
			if (!this.cancelled) {
				device.startListening();
				device.setEnableBroadcast(true);
				this.room.setDevice(device);
				ChatMIDlet.instance.setCurrent(this.room);
			}
		}

		public void setCancelled() {
			this.cancelled = true;
		}
}
```

D.7 net.java.dev.marge.util.ImageUtil

```
/*
 * Marge, Java Bluetooth Framework
 * Copyright (C) 2006  Project Marge
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-
1301  USA
 *
 * owner@marge.dev.java.net
 * http://marge.dev.java.net
 */

package net.java.dev.marge.util;

import java.io.IOException;

import javax.microedition.lcdui.Image;
```

```java
public class ImageUtil {

	public static Image loadImage(String filename) {
		Image i = null;
		try {
			i = Image.createImage(filename);
		} catch (IOException e) {
			e.printStackTrace();
		}
		return i;
	}
}
```