UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

DESENVOLVIMENTO SYMBIAN NA PLATAFORMA SERIE 60

RENATO FARIA IIDA

ORIENTADOR: DR. LEONARDO R. A. X. DE MENEZES

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGENE.DM – 273 A/06
BRASÍLIA/DF: SETEMBRO - 2006

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

DESENVOLVIMENTO SYMBIAN NA PLATAFORMA SERIE 60

RENATO FARIA IIDA

DISSERTAÇÃO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADA POR:

____

Prof. Leonardo R. A. X. de Menezes, Dr. (ENE-UnB)
(Orientador)

____

Prof. Paulo Henrique Portela de Carvalho, Dr. (ENE-UnB)
(Examinador Interno)

____

Prof. Marcello Luiz Rodrigues de Campos, Dr. (COPPE-UFRJ)
(Examinador Externo)

BRASÍLIA/DF, 29 DE SETEMBRO DE 2006

FICHA CATALOGRÁFICA

IIDA, RENATO FARIA
Desenvolvimento Symbian na plataforma serie 60 [Distrito Federal] 2006.
xxii, 116p., 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2006). Dissertação de Mestrado – Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica.

1. SYMBIAN	2. Celular
3. LBS	4. Desenvolvimento
I. ENE/FT/UNB.	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

IIDA, RENATO FARIA (2006). Desenvolvimento Symbian na plataforma serie 60. Dissertação de Mestrado, Publicação PPGENE.DM-273A/06, Departamento de Engenharia Elétrica, Universidade de Brasília , Brasília , DF, 116p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Renato Faria Iida.
TÍTULO: Desenvolvimento Symbian na plataforma serie 60.

GRAU: Mestre ANO: 2006

É concedida à Universidade de Brasília permissão para reproduzir cópias desta dissertação de mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desta dissertação de mestrado pode ser reproduzida sem a autorização por escrito do autor.

____

Renato Faria Iida
Campus Universitário Darcy Ribeiro, Faculdade de Tecnologia.
CEP 70910-900 – Brasília – DF - Brasil

DEDICATÓRIA

Dedico esta dissertação de mestrado às duas pessoas mais importantes da minha vida, meus pais.
Minha mãe por sempre me ajudar e cuidar de mim. Além de sempre me mandar estudar e me motivar a terminar este trabalho e muitos outros durante a minha vida.
Meu pai por ser um exemplo de professor e de engenheiro. Manteve os meus objetivos de vida em foco, sempre criando novos patamares a serem alcançados.

AGRADECIMENTOS

Várias pessoas me ajudaram a concluir esse trabalho.

Primeiramente, agradeço à paciência e ao apoio da Giza, pois tive que ficar várias noites e finais de semana para terminar esse trabalho sem poder dar atenção a ela.

Agradeço ao Eduardo Peixoto por ser uma pessoa chave que me ajudou a criar grande parte desse material.

Agradeço ao Alex por ajudar a criar esse texto no formato correto e sempre me motivar a seguir em frente.

Agradeço ao Xaxim e todos do grupo dele de projeto final pelas idéias no desenvolvimento de aplicações baseadas em localização.

Agradeço ao professor Itiro pela revisão de vários pontos desse texto.

Agradeço ao LEMOM e ao professor Paulo, que gentilmente cedeu o celular Nokia 6600 que foi usado nos testes desse projeto.

Ao meu orientador, Leonardo, por ter me guiado na direção correta e por ter me dado um voto de confiança no meu trabalho.

RESUMO

Desenvolvimento Symbian na plataforma serie 60

Autor: Renato Faria Iida

Orientador: Dr. Leonardo R. A. X. De Menezes

Programa de Pós-Graduação em Engenharia Elétrica

Brasília, Setembro de 2006

Com o rápido avanço da tecnologia dos terminais celulares e a integração de funções de *handhelds* aos celulares, criam-se novas possibilidades de aplicações móveis. Atualmente existem várias linhas de desenvolvimento. As três principais linguagens são:

- Symbian C++
- Java 2 Micro Edition
- BREW

O foco desse trabalho é mostrar o ciclo de desenvolvimento do Symbian C++ aplicado para aplicações baseadas em localização. O primeiro passo é entender a origem e as características básicas da linguagem. Após isso, é necessário entender como a interface gráfica é organizada e implementada. Por necessitar de usar bibliotecas assíncronas, é apresentado como funciona a multitarefa dentro da linguagem. A comunicação com o terminal GPS é feita por bluetooth. Portanto, a implementação dessa comunicação é apresentada. Outro aspecto é um estudo comparativo entre os métodos de obter informações do celular.

A metodologia usada para apresentar esses pontos é mostrar trechos de códigos integrados ao texto para facilitar a associação da parte teórica com a implementação. Além disso, vários exemplos, cada um com uma função específica, ajudam a ilustrar os conceitos dos capítulos. Ao final, um exemplo apresenta a integração de todos os conhecimentos anteriores e mostra a comunicação do celular Nokia 6600 [27] com o GPS Holux GPSlim236 [26].

ABSTRACT OU RÉSUMÉ

Development Symbian in serie 60 platform

Author: Renato Faria Iida

Supervisor: Dr. Leonardo R. A. X. De Menezes

Programa de Pós-Graduação em Engenharia Elétrica

Brasília, September de 2006

The fast development of the cell phones and the integration of features of handhelds in them created new opportunities for mobile applications. The main development languages are:

- Symbian C++
- BREW
- Java 2 Micro Edition

The main focus in this work is the development cycle of Symbian C++ applied to location based services. The first step is to understand the origin and basics characteristics of the language. After that, it is necessary to understand how the graphic interface is organized and implemented. Asynchronous calls are need, so it is explained how to implement multitask. The communication with the GPS device is made by Bluetooth. Because of this, the implementation of this communication is shown. Another aspect is comparison between the methods to get the phone information.

The methodology is to show samples codes between the texts to facilitate the learning process. In addition, application examples will be created to show the features. In the end, one example that shows all the features integrated.

SUMÁRIO

1 INTRODUÇÃO 1
1.1 MOTIVAÇÃO 2
1.2 OBJETIVOS 2
1.3 PRÉ-REQUISITOS E METODOLOGIA 3
1.4 ESTRUTURA DOS CAPITULOS 3
2 LOCATION BASED SERVICES 5
2.1 MÉTODOS DE LOCALIZACÃO 5
2.2 GPS 6
2.2.1 Partes do sistema GPS 7
2.2.2 Protocolo NMEA 7
2.3 EXEMPLOS DE APLICACOES EM LBS 13
2.4 CONCLUSÃO DO CAPITULO 13
3 SYMBIAN 14
3.1 SYMBIAN OS 14
3.2 SYMBIAN C++ 15
3.3 DEFINIÇÃO DE COMPONENTES 17
3.4 ESPECIFICAÇÕES DE PROJETO 17
3.5 UID – UNIQUE IDENTIFIER NUMBER 18
3.6 CONSTRUINDO O CÓDIGO 19
3.7 EXPLICANDO O CÓDIGO 20
3.8 COMPILAR E CONSTRUIR O CÓDIGO EXECUTÁVEL E ARQUIVO DE INSTALAÇÃO 22
3.9 VERIFICAÇÃO DE BALANÇO DA HEAP 24
3.10 GERENCIAMENTO DE ERROS 25
3.11 FUNÇÕES L 26
3.12 CLEANUP STACK (PILHA DE LIMPEZA) 26
3.13 CONSTRUTORES DE SEGUNDA-FASE 27
3.14 CONCLUSÃO DO CAPITULO 29
4 INTERFACES GRAFICAS EM SYMBIAN C++ 30
4.1 ARQUITETURA DE UMA APLICAÇÃO 30
4.2 O FRAMEWORK DO SISTEMA OPERACIONAL 31

4.2.1 Classe Application 33
4.2.2 Classe Document 34
4.2.3 Classe AppUI 35
4.2.4 Classe AppView 35
4.3 INICIAÇÃO DE UMA APLICAÇÃO 35
4.4 PROJETO DA ARQUITETURA DA APLICAÇÃO 38
4.5 ARQUITETURA TRADICIONAL 39
4.6 ARQUIVO DE RECURSOS 41
4.6.1 Definir um menu utilizando o Arquivo de Recurso 42
4.6.2 Manuseando os Comandos 43
4.6.3 Menus Dinâmicos 44
4.6.4 String no Arquivo de Recurso e Internacionalização 45
4.6.5 Função para Mudar o Texto 46
4.7 GERANDO O PROJETO 46
4.8 CONCLUSÃO DO CAPITULO 47
5 MULTI TAREFA 48
5.1 MULTITASKING COOPERATIVO 48
5.2 ACTIVE SCHEDULER 48
5.3 ACTIVE OBJECTS 48
5.3.1 Implementando um objeto Ativo 50
5.4 CONCLUSÃO DO CAPITULO 50
6 COMUNICAÇÃO EM SYMBIAN C++ 51
6.1 TIPOS DE COMUNICAÇÃO 51
6.2 BLUETOOTH 51
6.2.1 Anúncio e descoberta de serviços em Bluetooth 52
6.2.2 Procurando e conectando a um dispositivo bluetooth 54
6.2.3 Comunicação *socket* entre os dispositivos 58
6.3 CONCLUSÃO DO CAPITULO 59
7 INFORMAÇÕES SOBRE O DISPOSITIVO 60
7.1 INFORMAÇÕES DO TELEFONE USANDO RBASICGSMPHONE... 60
7.1.1 Vantagens do método RBasicGsmPhone 62
7.1.2 Desvantagens do método RBasicGsmPhone 62
7.2 INFORMAÇÕES DO TELEFONE USANDO MOBINFO 62

7.2.1 Vantagens da MobInfo 63
7.2.2 Desvantagens da MobInfo 64
7.3 CONCLUSÃO DO CAPITULO 64
8 APLICAÇÃO FINAL 65
8.1 INTERFACE DO PROGRAMA 65
8.2 MENU 66
8.3 FUNCIONALIDADES 67
8.3.1 Obter os dados GPS via bluetooth 67
8.3.2 Obter dados do telefone via MobInfo 70
8.3.3 Obter dados do telefone via RBasePhone 70
9 CONCLUSÃO 71
9.1 RESULTADOS 71
9.2 DIFICICULDADES SOBRE DESENVOLVIMENTO SYMBIAN 71
9.3 MELHORIAS E NOVOS PROJETOS 72
REFERÊNCIAS BIBLIOGRÁFICAS 73
APÊNDICES
APÊNDICE A - CD DE INSTALAÇÃO 77
APÊNDICE B -EXEMPLOS 78
APÊNDICE C - AMBIENTE DE DESENVOLVIMENTO 79

LISTA DE TABELAS

Tabela 2-1 Requisitos de Performance em LBS[29] 5
Tabela 2-2 Definição dos campos da mensagem GPGGA. 9
Tabela 2-3 Definição dos campos da mensagem GPVTG. 10
Tabela 2-4 Definição dos campos da mensagem GPGSV. 11
Tabela 2-5 Definição dos campos da mensagem GPRMC. 12
Tabela 2-6 Tipos de serviço de localização 13
Tabela 3-1 Principais tipos básicos no Symbian C++ 16
Tabela 3-2 Principais tipos básicos no Symbian C++ 18

LISTA DE FIGURAS

Figura 3.1 Simulação da aplicação "*HelloText*" 23
Figura 4.1 - Diagramas das classes núcleo 32
Figura 4.2 Seqüência de inicialização de uma aplicação. 36
Figura 4.3 Tela Inicial do Exemplo. 40
Figura 4.4 Menu da Tela Inicial do Exemplo. 40
Figura 4.5 Tela Final do Exemplo. 40
Figura 4.6 Menu da Tela Final do Exemplo. 41
Figura 8.1 Diagrama dos elementos da aplicação final. 65
Figura 8.2 Exemplo de interface. 65
Figura 8.3 Menu Inicial. 67
Figura 8.4 Tela de busca de dispositivos. 68
Figura 8.5 Conectado ao GPSSlim236. 68
Figura 8.6 Novo Menu com o GPS conectado. 69
Figura 8.7 Dados do GPS 69
Figura 8.8 Resultado do MobInfo 70
Figura 8.9 Resultado do RBasePhone. 70
Figura 9.1 Criando um novo projeto 80
Figura 9.2 - Tipos do projeto móvel. 81
Figura 9.3 – Selecionar o arquivo bld.inf para importar um projeto 81
Figura 9.4 – Novo Projeto. 82
Figura 9.5 Instalação do JRE 1.5 update 6 83
Figura 9.6 Escolha de pacote no ActivePearl 84
Figura 9.7 Opções do ActivePearl. 84
Figura 9.8 Aviso de pré-requisitos no S60 SDK. 85
Figura 9.9 Tipo de instalação do S60 SDK. 86
Figura 9.10 Instalação de pacotes do SDK. 86
Figura 9.11 Definição do *workspace* 87
Figura 9.12 Criando um novo projeto 88
Figura 9.13 Nome do novo Projeto. 88
Figura 9.14 Tipos de Projetos 89
Figura 9.15 Escolha do SDK do projeto. 89

Figura 9.16 Importar projeto. 90
Figura 9.17 Escolhendo o bld.inf para importar. 91
Figura 9.18 *Parse* do arquivo *.mmp*... 92
Figura 9.19 SDK usado pelo projeto importado.. 92

Índice de Listagens

Listagem 3.1 Conteúdo do arquivo bld.inf. 17

Listagem 3.2 Conteúdo do arquivo hellotext.mmp 18

Listagem 3.3 Código fonte do programa – hellotext.cpp. 20

Listagem 3.4 - Utilizando Traps. 25

Listagem 3.5 - Utilizando a *Cleanup Stack* – parte 1. 26

Listagem 3.6 Utilizando a *Cleanup Stack* – parte 2. 27

Listagem 3.7 - Utilizando Construtores de segunda fase – parte 1. 28

Listagem 3.8 - Utilizando Construtores de segunda fase – parte 2 – newL(). 28

Listagem 3.9 - Utilizando o operador newL(). 28

Listagem 3.10 - Utilizando Construtores de segunda fase – parte 2 – newLC(). 29

Listagem 3.11 - Utilizando o operador newLC(). 29

Listagem 4.1 Implementação comum da classe Application 33

Listagem 4.2 Implementação comum da classe Document. 34

Listagem 4.3-Métodos para inicalização da aplicação. 36

Listagem 4.4-Método para identificar a aplicação. Este fragmento encontra-se na classe Application 37

Listagem 4.5 - Método para instanciar a classe Document. Este fragmento encontra-se na classe Application. 37

Listagem 4.6 Método para instanciar a classe AppUI. Este fragmento encontra-se na classe Document. 37

Listagem 4.7 Cabeçalho do Arquivo de Recurso (HelloWorld.rss). 41

Listagem 4.8 Declaração do Menu no Arquivo de Recurso (HelloWorld.rss) 42

Listagem 4.9 - Definição dos itens do menu (HelloWorld.rss). 42

Listagem 4.10 - Associação do menu a tela do usuário (HelloWorld.rss). 42

Listagem 4.11 - Arquivo de definição dos IDs dos comandos do menu (HelloWorld.hrh). 43

Listagem 4.12 - Código para manipular os eventos do menu. (HelloWorldAppUi.cpp). 44

Listagem 4.13 - Código para criar um menu dinâmico (HelloWorldAppUi.cpp) 44

Listagem 4.14 - Arquivo com as *strings* para cada língua (HelloWorld.loc). 45

Listagem 4.15 - Texto da língua inglesa para a aplicação (HelloWorld.l01)............... 45
Listagem 4.16 - Inserindo uma string como recurso (HelloWorld.rss)........................ 45
Listagem 4.17 - Função para mudar o texto na tela (HelloWorldContainer.cpp)... 46
Listagem 4.18 - Código do arquivo bld.inf.. 46
Listagem 4.19 - Código do arquivo HelloWorld.mmp. 46
Listagem 5.1 – Funções do CAtive 49
Listagem 6.1 Conectando na base de dados (SDD).. 52
Listagem 6.2 criando uma nova entrada no SDD.. 53
Listagem 6.3 Criando o servidor *socket*.. 53
Listagem 6.4 Obtendo a porta disponível. 53
Listagem 6.5 Configurando a entrada do SDD. 53
Listagem 6.6 Configurando o servidor *socket* para novas conexões. 54
Listagem 6.7 Busca de dispositivos bluetooth. 55
Listagem 6.8 Iniciando uma procura pelo serviço dentro de MSdpAgentNotifier.... 56
Listagem 6.9 Definição do método NextRecordRequestCompletL.................... 56
Listagem 6.10 Definição do método AttributeRequestResultL.............................. 57
Listagem 6.11 Definição do método VisitAttributeValueL..................................... 58
Listagem 6.12 Manipulação do *socket* no servidor. 59
Listagem 6.13 Criando o *socket* no cliente com o servidor. 59
Listagem 7.1 Inicialização da Estrutura RBasicGsmPhone. 61
Listagem 7.2 Obtenção do cell id 61
Listagem 7.3 Obtendo o IMEI... 61
Listagem 7.4 Trecho de “mobinfotypes.h”. 62
Listagem 7.5 Obtendo IMEI e *Cell Id* via mobinfo API. 63
Listagem 8.1 Funções da interface MLog 66
Listagem 8.2 Definição do menu no arquivo .rss 66
Listagem 8.3 Implementação do DynInitMenuPaneL... 66

LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIAÇÕES.

3GPP - *3rd generation partnership project* – Um acordo de colaboração cujo escopo é firmar uma especificação globalmente aplicável para os serviços de terceira geração.

A

A5/1 – Algoritmo de criptografia em cadeia usada para privacidade de voz na tecnologia GSM

A5/2 – Algoritmo de criptografia em cadeia usada para privacidade de voz na tecnologia GSM, porém mais fraco que a A5/1.

AES - *advanced encryption standard* – Algoritmo de criptografia em bloco de tamanho fixo e chave de 128, 192 ou 256 bits.

AJAX – *asynchronous javasript and xml* – Técnica de desenvolvimento web para criação de aplicações web interativas.

API – *application programmer interface* – Conjunto de rotinas e padrões estabelecidos por um *software* para utilização de suas funcionalidades.

ASCII – *american standard code for information interchange* – Codificação de caracteres baseado no alfabeto inglês.

AC - *Authentication Centre* - Servidor para autenticar o cartão SIM no sistema GSM

B

BSC - *Base station Controller* – Parte de controle da interface aérea do sistema GSM.

BTS – *Base station Transceiver* – Parte de transmissão e recepção na interface aérea do sistema GSM

C

CDC – *connected device configuration* – Configuração da J2ME para dispositivos sem limitações de energia.

CDMA – *code division multiple access* – Tecnologia de acesso que realiza multiplexação do canal pelo uso de diferentes códigos.

CLDC – *connection limited device configuration* – Configuração da J2ME para dispositivos com limitações de energia.

CR – *carriage return* – Caractere de controle em código ASCII para fazer o cursor retornar para o início da linha em um documento de texto. Seu valor decimal é 13.

D

DES – *data encryption standard* – Algoritmo de criptografia em bloco de tamanho fixo e chave de 56 bits.

DH – Diffie-Helman – Algoritmo de troca de chaves que serão usadas em criptografia.

DNS – *domain name system* – Sistema para gerenciamento hierárquico e distribuído de nomes de nós na rede.

DSA – *digital signature algorithm* – Algoritmo assimétrico de criptografia para criação de assinaturas digitais.

E

EDGE – *enhanced data rates for global evolution* – Tecnologia que adiciona serviços de terceira geração à rede GSM.

ERB – Estação Rádio-base

ETSI - *european telecommunications standards institute* – Organização padronizadora européia sem fins lucrativos da indústria de telecomunicações.

F

FTP – *file transfer protocol* – Protocolo para transferência de arquivos na rede.

FM – *frequency modulation* – Modulação em freqüência.

FSK – *frequency shift keying* – Modulação por chaveamento de freqüência.

G

GCF – *generic connection framework* **–** Estrutura básica em Java para operações de entrada e saída de dados em dispositivos limitados.

GIAC – *general inquiry access code* – Modo de procura de dispositivos Bluetooth®.

GMSK – *gaussian minimum shift keying* – tipo de FSK com fase continua.

GPINF – *gps information* **–** Protocolo desenvolvido neste projeto para comunicação entre os dispositivos do sistema.

GPL – *general public license* – Licença de software livre mais popular.

GPRS – *general packet radio service* – Tecnologia de comutação de pacotes que aumenta a taxa de transferência de dados em redes GSM.

GPS – *global positioning sytem* – Sistema de posicionamento global.

GSM – *global system for mobile communications* – Padrão europeu para redes de telefones celulares digitais.

H

HTML – *hyper-text markup language* – Linguagem de marcação utilizada para produzir páginas na internet.

HTTP – *hypertext transfer protocol* – Protocolo da camada de aplicação para transferência de hiper-mídia (imagens, sons e textos).

HTTPS – *hypertext transfer protocol secure* – Protocolo da camada de aplicação para transferência segura de hiper-mídia (imagens, sons e textos).

HLR – *Home Location Register* - o registro permanente dos usuários do sistema GSM

I

ICMP – *internet control message protocol* – Protocolo da camada de rede integrante do protocolo IP. Fornece relatórios de erros ao remetente dos dados.

IDEA - *international data encryption algorithm* – Algoritmo de criptografia em bloco.

IP – *internet protocol* – Protocolo para internet responsável pela identificação de nós e roteamento de pacotes.

IMSI - *International Mobile subscriber Identity* – Código identificador do usuário celular na rede GSM, armazenado no Sim card e na HLR.

IMEI - *International Mobile Equipment Identity* – Número de serie do aparelho celular.

J

J2EE – *Java 2, enterprise edition* – Ambiente de desenvolvimento Java para servidores.

J2ME – *Java 2, micro edition* – Ambiente de desenvolvimento Java para dispositivos limitados.

J2SE – *Java 2, standard edition* – Ambiente de desenvolvimento Java para *desktops*.

JDBC – *Java database connectivity* – API Java de acesso ao banco de dados.

JSP – *java server pages* - Tecnologia para desenvolvimento de aplicações WEB.

JSR – *Java specification request* – Descrição de uma especificação proposta a Comunicade Java para a criação de uma API.

K

kbps – kilobits por segundo – unidade de transmissão de dados igual a 1000 bits por segundo.

L

LAN – *local area network* – Rede local de computadores.

LF – *line feed* – Caractere de controle em código ASCII usado para fazer o cursor ir para a linha de baixo em um documento de texto. Seu valor em decimal é 10.

M

MD5 – *message digest algorithm 5* – Algoritmo de função *hash* de 128 bits.

MIDP – *mobile information device protocol* – Perfil da J2ME

MMC – *multi-media card* – Pequeno cartão de memória para transportar dados entre diferentes dispositivos.

MVC – *model-view-controller* – Padrão de projeto de desenvolvimento de software.

MSC – *Mobile Switching Centre* – Elemento de controle da rede GSM.

MSISDN - *Mobile Subscriber International* – Número do celular

N

NAVSTAR – *navigation system using time and ranging* – Sistema de posicionamento baseado em satélites.

NMEA – *national marines eletronic association* – Organização profissional que define e mantém os formatos seriais padrões utilizados por equipamentos eletrônicos de navegação marítima e interface de computadores.

P

PAN – *personal area network* – Rede formada por dispositivos próximos uns dos outros.

PDA – *personal digital assistent* – Computador de dimensões reduzidas, portátil e com as funções de agenda, bloco de notas e lista de contatos.

PPP – *point-to-point protocol* – Interface para transporte de dados entre dois dispositivos de rede através de uma conexão física única.

PPS – *precise positioning service* – Sistema de posicionamento de uso militar.

R

RAM – *random access memory* – Memória de conteúdo volátil disponível para uso de aplicações e processos.

RC2, RC4 e RC5 – *Ron's code* – Algoritmos de criptografia em bloco desenvolvidos por Ron Rivest. Os blocos possuem tamanho fixo e as chaves, tamanho variável.

RMI – *remote method invocation* – Processo para invocação de métodos de objetos Java pertencentes a diferentes máquinas virtuais.

ROM – *read only memory* – Memória com dados gravados permanentemente destinados apenas para leitura.

RSA – Ron Rivest, Adi Shamir, Len Adleman – Algoritmo de criptografia com chave assimétrica.

S

SHA – *secure hash algorithm* – Algoritmo de função *hash*.

SIM – *subscriber identity module* – Modulo de identificação do assinante GSM.

SPS – *standard positioning service* – Sistema de posicionamento de uso civil.

SPP – *serial port profile* – Perfil que define o procedimento necessário para configurar conexões seriais.

SMS – *short message service* – Serviço disponível em telefones celulares (tele móveis) digitais que permite o envio de mensagens curtas entre estes equipamentos e entre outros dispositivos de mão como palm e handheld, e até entre telefones fixos. (Ver bibliografia)

SSL – *secure sockets layer* – Protocolo criptográfico que provê comunicação segura na Internet. Sucedido por TLS.

T

TDL – *tasknet device languagem* – Linguagem para programação dinâmica de dispositivos no sistema TaskNet.

TDMA – *time division multiple access* – Esquema de multiplexação por divisão de tempo.

TLS – *transport layer security* - Protocolo criptográfico que provê comunicação segura na Internet, sucessor do SSL.

TCP – *transmission control protocol* – Protocolo da camada de transporte que estabelece a conexão e assegura sua confiabilidade.

U

UDP – *user datagram protocol* – Protocolo da camada de transporte que fornece acesso direto ao serviço de entrega de datagramas.

UI API – *user interface api* – API com classes para criação de uma interface, por vezes gráfica, entre a aplicação e o usuário.

URL – *universal resource locator* – Endereço de um recurso disponível em uma rede.

USB – *universal serial bus* – Conexão *plug-and-play* entre dispositivos e periféricos sem necessidade de desligar o dispositivo.

V

VLR – *Visitor Location Register* (VLR) - base de dados temporário como dados usários conectados na rede GSM.

W

W3C – *world wide web consortium* – Consórcio internacional de desenvolvimento de padrões para WWW.

WWW – *world wide web* – Espaço global de informações onde as pessoas podem ler e escrever através de computadores conectados à Internet.

X

XML – *extensible markup language* – Recomendação W3C.

1 INTRODUÇÃO

No atual cenário do desenvolvimento para celulares, existem três principais plataformas de desenvolvimento de software para um celular:

- Java 2 Micro Edition
- BREW
- Symbian

O Java 2 Micro Edition (J2ME) foi desenvolvido para ser utilizado por celulares, *pagers*, PDAs, dentre outros dispositivos de acesso e recursos restritos como automóveis ou *settop box* de TV Digital. Devido às grandes diferenças entre eles, foram criadas as configurações, que definem subgrupos reunindo aqueles dispositivos semelhantes ente si. Essas configurações especificam as características da linguagem Java, da Java *Virtual Machine* e do conjunto de bibliotecas e API`s suportadas.

Atualmente, existem dois tipos de configurações: *Connected Device Configuration* (CDC) e *Connected Limited Device Configuration* (CLDC). O CDC é utilizado por dispositivos mais poderosos, com menor restrição no consumo de energia, como Internet TVs ou sistemas de navegação de carros. O CLDC, por outro lado, é utilizado por dispositivos menos poderosos, como celulares, *palms*, PDAs, com recursos limitados com relação a consumo de energia, quantidade de memória, entre outros. Ambas as configurações presumem que o aparelho esteja conectado a algum tipo de rede (que é normalmente sem fio e com pequena velocidade de transmissão, no caso do CLDC). Tanto a CDC quanto a CLDC possuem sua própria Java *Virtual Machine*, sendo denominadas a KVM(*Kilo Virtual Machine)* e a CVM *(Compact Virtual Machine*).

Para celulares o perfil utilizado é o MIDP, que possui atualmente duas versões: 1.0 e 2.0. Cada uma destas versões inclui um conjunto de APIs para auxiliar os desenvolvedores. Mas essa máquina virtual tem muitas limitações e vários recursos do aparelho podem não estar disponíveis.

BREW significa *Binary Runtime Environment for Wireless* e não é apenas um conjunto de bibliotecas para o desenvolvimento de aplicações, mas também um meio de comercialização e entrega destas para os usuários finais. BREW foi desenvolvido pela Qualcomm, que supervisiona todo o desenvolvimento. Atualmente, o BREW é a linguagem mais usada para o desenvolvimento em celulares CDMAone. É uma

linguagem baseada em C, mas com várias *macros* de programação para acessar os diversos recursos do celular.

O Symbian é um sistema operacional de celulares que tem uma linguagem própria para desenvolvimento de aplicativos chamado Symbian C++. Detalhes desse sistema e a linguagem serão mostrados no decorer desse trabalho.

1.1 MOTIVAÇÃO

Durante o projeto Inova Mobile[32], várias aplicações baseadas em Java foram desenvolvidas usando o MIDP 1.0. Entretanto, várias limitações foram encontradas. Apesar de existir conexão bluetooth e informações como IMEI e *cell id*, esse recurso e as informações não estavam disponíveis para *virtual machine*. O Symbian OS oferecia esses recursos. O desenvolvimento em Symbian é feito em C++. Isso acarreta uma maior velocidade, mas um desenvolvimento mais complexo para ser entendido e desenvolvido. Esse cenário determinou os objetivos mostrados no item 1.2.

1.2 OBJETIVOS

Os objetivos desse projeto são:

- Mostrar uma introdução ao desenvolvimento Symbian para difundir a linguagem e motivar a formação de novos desenvolvedores.
- Apresentar as ferramentas necessárias para desenvolver aplicativos para Symbian OS para executar e testar os exemplos mostrados nesse trabalho.
- Mostrar, através de trechos de código e exemplos, as principais funcionalidades de linguagem de desenvolvimento para Symbian OS. Essa é uma forma fácil e prática de aprender a criar projetos em uma linguagem.
- Implementar a integração via bluetooth de um GPS com o celular rodando Symbian OS. Uma limitação do GSM para os sistemas CDMA é a que o sistema CDMA dispõe de um sistema de localização denominado GPSOne integrado ao terminal do usuário. Utilizando esse hardware externo, tenta-se minimizar essa limitação.

1.3 PRÉ-REQUISITOS E METODOLOGIA

O conhecimento de orientação objeto e da linguagem C++ são importantes para entender os códigos de exemplos mostrados do decorrer dos próximos capítulos. Conceitos do sistema GSM são recomendados para entender alguns parâmetros mostrados no capítulo 7.

A metodologia é mostrar trechos de códigos integrados ao texto para facilitar a associação da parte teórica com a implementação. Os trechos de código têm uma formatação diferenciada e tem nome de listagem. Quando um termo técnico de programação é apresentado no texto, é usado esse `formato`. Para termos em ingles é usado o *formato*.

1.4 ESTRUTURA DOS CAPITULOS

Essa dissertação estrutura-se em oito capítulos.

O capítulo 1 apresenta uma introdução, mostrando a motivação e os objetivos do trabalho.

O capítulo 2 apresenta o conceito de serviços baseados em localização. Mostra alguns tipos de métodos para implementar isso em uma rede celular com ênfase no sistema GPS e seus comandos.

O capítulo 3 apresenta uma introdução ao Symbian OS e à linguagem Symbian C++, além de mostrar os principais conceitos da linguagem para entender os códigos desse capítulo e dos posteriores.

O capítulo 4 apresenta a interface gráfica do Symbian OS. Mostra três formas de criar a arquitetura da interface gráfica, além de explicar como criar um arquivo de recursos para definir o projeto da interface.

O capítulo 5 apresenta implementação de multitarefa em Symbian OS. Isso é feito utilizando um padrão denominado de objetos ativos que utiliza um sistema cooperativo. Esse capítulo é importante para entender como implementar as chamadas assíncronas de bibliotecas aprentadas nos capítulos 6 e 7.

O capítulo 6 apresenta a comunicação em Symbian OS com ênfase em bluetooth. Mostra como se deve registrar os serviços bluetooth em um servidor e como descobrir e conectar através de um cliente. Esse tipo de conexão será usado para obter as informações de um dispositivo GPS externo.

O capítulo 7 apresenta formas de obter informações sobre o celular dentro do Symbian OS, analisando as vantagens, desvantagens e quais versões de Symbian OS suportam esses métodos.

O capítulo 8 apresenta o exemplo final integrando os conhecimentos dos capítulos anteriores. Mostra cada funcionalidade do programa com *screenshots* do celular de teste Nokia 6600 [27].

2 LOCATION BASED SERVICES

Os serviços baseados em localização ou *Location Based Services (LBS)* são aplicações móveis com a característica de utilizar a posição atual ou armazenada do móvel para prover novos tipos de serviços para usuário [24]. Um dos primeiros serviços foi E911[28].Era uma determinação do FCC que foi dividida em duas fases. A primeira fase é que as ligações para o 911 deveriam conter a célula (*cell id*) e o número do celular de origem no ano de 1998. A segunda é aumentar a precisão da chamada de origem para 100 metros no ano de 2001. Na seção 2.1, várias formas de implementar essa localização na rede celular são mostradas. Um enfoque no método GPS é apresentado, pois será o método utilizado na aplicação do capitulo 8. Além disso, um detalhamento dos comandos NMEA do hardware utilizado nesse projeto é mostrado no final do capítulo.

2.1 MÉTODOS DE LOCALIZACÃO

Existem vários métodos de localização [29]. O critério de analise dos métodos será a desempenho de cada um. Esta performance é medida pelos critérios da Tabela 2-1.

Tabela 2-1 Requisitos de Performance em LBS[29]

Requisito de performance	Descrição
Consistência	Os resultados devem ser consistentes em diferente locais de medição. Ou seja, um resultado não pode variar de 100 metros para 2 quilômetros só pela mudança de região.
Acurácia	A acurácia é a diferença da posição real com a fornecida pelo sistema.
Tempo de para iniciar	O tempo para iniciar o sistema e fornecer a primeira medição. O termo comum para isso é TTFF (*time-to-first-fix*)

A descrição e análise de cada método são mostradas abaixo.

- Célula de Origem (COO) – utiliza o *cell id* onde o celular que deseja encontrar está registrado. Esse é o método mais simples de localização. O custo é muito baixo, pois as informações de *cell id* do sistema estão disponíveis e a grande maioria dos celulares programáveis tem acesso a essa informação para o desenvolvedor (veja no Capítulo 7). Isso torna o uso desse sistema praticamente imediato. Infelizmente, esse sistema é muito impreciso e extremamente dependente do tamanho da célula, que na rede GSM típica varia de 2 km até 20 km.
- Observação da diferença de tempo avançada (E-OTD) – O celular (observador) envia dados para as estações-base próximas e mede a diferença de tempo das respostas. Deve-se analisar o tempo de pelo menos três estações-base para ter uma localização 2-D. Para implementar isso na rede GSM, é necessário instalar um novo hardware denominado LMU (*Location Measurement Unit*) nas estações-base e mudança no terminal móvel. A acurácia de 100 a 500 metros. O tempo de TTFF é por volta de 5 segundos.
- GPS – sistema global de localização. Utiliza uma rede de satélites para obter a localização do terminal móvel. É considerado o sistema mais acurado de localização variando de 5 a 50 metros. Além de obter a localização em 3-D. Para isso, é necessário o terminal estar conectado a quatro ou mais satélites para obter dados confiáveis. Isso dificulta a implementação desse serviço em ambientes fechados. Os celulares GSM precisam de um hardware externo para obter os dados dessa rede de satélites. Nesse projeto, é utilizado o GPSSlim236[26].

2.2 GPS

O sistema global de posicionamento (GPS) foi criado pelos militares dos Estados Unidos para que, em qualquer lugar do mundo, os soldados americanos pudessem ser localizados com uma margem de erro de 5 a 50 metros. Uma característica adicional é que o terminal para o soldado seria somente de recepção. A finalidade disso é evitar a detecção do mesmo por unidades inimigas[8].

2.2.1 Partes do sistema GPS

O sistema GPS é dividido em três partes que serão descritas a seguir

- Parte Espacial
- Parte de Controle
- Parte do Usuário

2.2.1.1 Parte de Controle

A parte de controle do sistema é formada por uma estação central (MCS) e três antenas terrestres. A função básica é coletar informação da rede satelital e fazer ajustes nos dados de posicionamento e sincronismo.

2.2.1.2 Parte Espacial

A parte espacial do sistema é composta de 24 satélites sendo 21 operacionais e três reservas. A órbita desses satélites coloca-se em seis planos espaçados de 60 graus entre eles tendo três ou quatro satélites em cada plano. Eles estão a 20200 km de distancia da superfície terrestre.

2.2.1.3 Parte do Usuário

O aparelho da parte do usuário utiliza uma técnica de trilaterização. Basicamente, ela mede a distância dos lados do triângulo formado entre o ponto desconhecido e dois ou mais pontos conhecidos.

O sistema precisa de três satélites para determinar sua posição no espaço. Contudo, geralmente usa-se um quarto satélite. Esse último satélite é necessário por causa do receptor do usuário que é pouco preciso em relação ao dos satélites que usam quatro relógios atômicos. Então, com essa referência extra, os erros de sincronismo são minimizados. Para serem úteis, esses dados devem ser formatados para o usuário final. A norma mais comum para a formatação dos dados da rede GPS é o NMEA 183 que será mostrado abaixo.

2.2.2 Protocolo NMEA

A *National Marine Electronics Association* (NMEA) criou um protocolo de comunicação chamado de NMEA 183. É um protocolo padrão para comunicação entre equipamentos marítimos onde existe um TALKER (elemento gerador de informação) e um conjunto de LISTENERS (escutam a informação do TALKER). Vários

equipamentos utilizam esse protocolo para troca de informação entre eles. A saída originalmente definida no protocolo é EIA-422, usando 4800 bps, 8 bits de dados, sem paridade e um bit de *stop* (8N1). Mas atualmente existe uma extensão para usar 38400 bps. As sentenças enviadas do TALKER para os LISTENERS são todas ASCII. Cada sentença começa com símbolo $ e seguido de um identificador do TALKER com duas letras. A norma engloba vários tipos de identificador, mas para essa documentação importa somente o identificador GP, que significa um aparelho de GPS. Após a indentificação do aparelho, tem um identificador de comandos com três letras. Os mais importantes e suas sintaxes serão mostradas no item 2.2.2.1. Além disso, o pacote termina com *carriage return linefeed* (<CR><LF>). Os campos dos comandos são delimitados por vírgula. Todas as vírgulas devem ser incluídas como marcadores. O *checksum* é adicionado opcionalmente.

2.2.2.1 Sentenças NMEA 183 importantes

A norma engloba vários comandos, mas o GPSSlim236 suporta apenas quatro comandos. Esses comandos são mostrados em detalhes abaixo. O entendimento é importante para obter as informações de localização mostrada para o usuário no exemplo do Capitulo 8.

GGA – Global Positioning System Fix Data – Latitude/Longitude: Essa sentença descreve a posição e o tempo fixos relacionados a um receptor GPS. A sintaxe é mostrada abaixo e a explicação de cada campo é mostrada na Tabela 2-2.

```
$GPGGA,hhmmss.dd,xxmm.dddd,<N|S>,yyymm.dddd,<E|W>,v,ss,d.d,h.h,M
,g.g,M,a.a,xxxx*hh<CR><LF>
```

Tabela 2-2 Definição dos campos da mensagem GPGGA.

hhmmss.dd	Tempo UTC hh – horas; mm – minutos. ss – segundos; dd – décimos de s.
xxmm.dddd	Latitude xx – Graus; mm – minutos dddd – décimos de minuto
<**N**|**S**>	Assume o valor N para Norte ou valor S para sul
yyymm.dddd	Longitude yyy – Graus ; mm – minutos dddd – décimos de minuto
<**E**|**W**>	Assume o valor E para Leste ou valor W para Oeste.
v	Indicador de Qualidade 0 – Invalido 1 – GPS 2- DGPS
ss	Número de satélites usados na posição fixa, 00-12. Comprimento fixo.
d.d	HDOP = Diluição Horizontal de Precisão.
h.h	Altitude (nível do mar, geóide).
M	Unidade da Altitude (metros).
g.g	Altitude em relação a elipsoide definida na norma WGS84
M	Unidade da Altitude (metros)
a.a	Campo Usado em DGPS
xxxxx	Campo Usado em DGPS

Exemplo do comando GPGGA gerado pelo GPSSlim236

```
$GPGGA,024556.000,1544.1610,S,04752.1164,W,1,07,1.2,1054.6,M,-
9.9,M,,0000*78
```

VTG – Course over ground: Descreve a velocidade do receptor. A sintaxe é mostrada abaixo e a explicação de cada campo é mostrada na Tabela 2-3.

```
$GPVTG,x.x,T,y.y,M,m.m,N,n.n,K
```

Tabela 2-3 Definição dos campos da mensagem GPVTG.

x.x	Variação em graus
T	Variação em relação ao norte real
y.y	Variação em graus
M	Variação magnética
m.m	Velocidade
N	Unidade em nos
n,n	Velocidade
K	Unidade de velocidade (Km/h)

Exemplo do comando GPGSV gerado pelo GPSSlim236

```
$GPVTG,054.7,T,034.4,M,005.5,N,010.2,K
```

GSV – Sattelites in View: Descreve o número de satélites em visada, o ID dos satélites (PRN), a elevação, o azimute e a relação sinal/ruído (SNR). Apenas quatro satélites por sentença, os outros satélites são enviados em uma próxima sentença. A sintaxe é mostrada abaixo e a explicação de cada campo é mostrada na Tabela 2-4.

```
$GPGSV,n,m,ss,xx,ee,aaa,cn,………… ,xx,ee,aaa,cn*hh<CR><LF>
```

Tabela 2-4 Definição dos campos da mensagem GPGSV.

n	Número total de mensagens, 1 a 9.
m	Número da mensagem, 1 a 9.
ss	Número total de satélites em visada
xx	Número do ID do satélite (PRN)
ee	Elevação do satélite, máximo 90 graus.
aaa	Azimute do satélite, grau verdadeiro, 00 a 359.
cn	SNR 00 – 99 dB - Hz 0 quando não está em movimento

Exemplo do comando GPGSV gerado pelo GPSSlim236

```
$GPGSV,2,1,08,11,58,230,35,23,40,274,20,01,37,161,38,19,30,333,30*7D
$GPGSV,2,2,08,20,24,219,23,13,13,297,20,03,13,002,29,16,09,038,35*7B
```

RMC – Recommended Minimum Specific GNSS Data: Descreve tempo, data, posição, curso e velocidade. A sintaxe é mostrada abaixo e a explicação de cada campo é mostrada na Tabela 2-5.

```
$GPRMC,hhmmss.dd,S,xxmm.dddd,<N|S>,yyymm.dddd,<E|W>,s.s,h.h,ddmm
yy,d.d,<E|W>*hh<CR><LF>
```

Tabela 2-5 Definição dos campos da mensagem GPRMC.

<table>
<tr><td>hhmmss.dd</td><td>Tempo UTC
hh – horas ; mm – minutos
ss – segundos ; dd – décimos de s.</td></tr>
<tr><td>S</td><td>Indicador de Status
A – Válido
V – Inválido</td></tr>
<tr><td>xxmm.dddd</td><td>Latitude
xx – Graus ; mm – minutos
dddd – décimos de minuto</td></tr>
<tr><td><**N**|**S**></td><td>Assume o valor N para Norte ou valor S para Sul</td></tr>
<tr><td>yyymm.dddd</td><td>Longitude
yyy – Graus ; mm – minutos
dddd – décimos de minuto</td></tr>
<tr><td><**E**|**W**></td><td>Assume o valor E para Leste ou valor W para Oeste.</td></tr>
<tr><td>s.s</td><td>Velocidade, nós</td></tr>
<tr><td>h.h</td><td>Título</td></tr>
<tr><td>ddmmyyy</td><td>Data
dd – dia ; mm – mês
yyy – ano</td></tr>
<tr><td>d.d</td><td>Variação Magnética</td></tr>
<tr><td><**E**|**W**></td><td>Declive. Assume o valor E para Leste ou valor W para Oeste.</td></tr>
</table>

Exemplo do comando GPRMC gerado pelo GPSSlim236

```
$GPRMC,024600.000,A,1544.1610,S,04752.1161,W,0.00,30.72,060606,,,A*50
```

2.3 EXEMPLOS DE APLICACOES EM LBS

Vários exemplos existem de serviços em localização, a Tabela 2-6 mostra esses exemplos divididos em categorias.

Tabela 2-6 Tipos de serviço de localização

Categorias de Serviços de Localização	Exemplos
Ativados por proximidade	• Cobrança sensível à posição • Propaganda por proximidade • Venda de bilhetes e entrada
Informação	• Relatório de transito • Logística
Rastreamento	• Gerenciamento de Frota • Rastreamento de animais • Rastreamento de pessoas
Assistência	• E911 • Notificação médica

2.4 CONCLUSÃO DO CAPITULO

Neste capítulo, a ênfase nos sistema de localização foi o GPS. Esse sistema foi escolhido para ser implementado no exemplo do Capítulo 8. O próximo capítulo inicia o enfoque mais prático do projeto com a introdução ao sistema Symbian OS e ao desenvolvimento na linguagem symbian C++ dentro desse sistema operacional.

3 SYMBIAN

Esse capítulo mostra a história e os principais conceitos para o desenvolvimento no sistema operacional Symbian. Além de, mostrar o primeiro exemplo prático usando interface de texto.

3.1 SYMBIAN OS

A história do Symbian começou em 1991, quando a Psion realizou o lançamento da primeira versão do sistema operacional EPOC para *handhelds*. Em 1997, foi lançado o Psion Series 5, o primeiro *handheld* a utilizar o sistema operacional EPOC32 (de 32 bits). Este aparelho chamou a atenção dos fabricantes de dispositivos celulares, que já vislumbravam a tendência desses aparelhos em agregar novas tecnologias e funcionalidades.

Em 1998 foi fundada a companhia Symbian, por um consórcio formado pela Nokia, Motorola, Ericsson e pela própria Psion. O objetivo da companhia era desenvolver uma versão do EPOC32 específica para aparelhos celulares – que foi chamado de Symbian OS.

O aparelho-alvo deste sistema operacional são os *smartphones* (aparelhos centrados em voz, com capacidade para dados) e *communicators* (aparelhos centrados em dados, com capacidade para voz). O primeiro *smartphone* com Symbian OS foi lançado em novembro de 2000, o Ericsson R380. Em junho de 2001 foi lançado o primeiro *communicator*, o Nokia 9210[31]. Outras empresas se juntaram ao Symbian mais tarde, adquirindo propriedade da companhia, enquanto a Motorola vendeu sua parte em 2003, após o lançamento do MPx200, aparelho com o sistema operacional Windows Mobile.

Muitas outras empresas possuem licença para fabricar aparelhos com este sistema operacional, e o próprio modelo de negócios possibilita isso. A empresa não cobra altas taxas de licenciamento. Para a produção, ela cobra uma taxa por unidade produzida. Com essa política dá chance a empresas menores lançarem seus produtos com competitividade com relação as maiores.

O Symbian OS foi projetado para rodar em telefones celulares, ou seja, é otimizado para a limitação de recursos. Em primeiro lugar, telefones celulares são dispositivos com recursos limitados em todos os sentidos. Por exemplo, baixo consumo de energia. Essa limitação se reflete em todos os outros elementos de hardware e de software.

O processador não pode ser tão rápido quanto à tecnologia permite, deve-se optar por um com o consumo reduzido.

As telas também são projetadas de forma que consuma o mínimo possível de energia.

O próprio sistema operacional precisa gerenciar essa energia limitada, além de estar preparado para uma eventual falta de energia, sem perda de dados.

Outro ponto de destaque aqui é o tipo de usuário ao qual se destina o sistema operacional. Usuários de PCs geralmente estão acostumados a reiniciar seus computadores todos os dias, ou ao menos regularmente. Um usuário de um telefone celular não tem este pensamento – é comum que sistemas rodando Symbian OS executem durante vários meses sem nunca reiniciar.

Para o desenvolvimento nesse sistema operacional foi desenvolvida uma linguagem adaptada do C++. A próxima seção vai mostrar uma introdução a essa nova linguagem.

3.2 SYMBIAN C++

Para explicar as principais peculiaridades do Symbian C++ utilizaremos a aplicação mais simples que o sistema operacional permite. É uma aplicação em console que escreve "*Hello World!*" na tela do dispositivo para plataforma Série 60.

Uma primeira diferença é que o Symbian C++ define seus próprios tipos primitivos (além de utilizar os do C++), e encoraja o desenvolvedor a utilizá-los. A razão para isso é que eles têm consistência no Symbian OS, tendo garantias de seu comportamento, independente da implementação (por exemplo, um TInt tem sempre 32 bits). Esses tipos primitivos podem ser vistos na Tabela 3-1.

Tabela 3-1 Principais tipos básicos no Symbian C++

Tipos Primitivos			Descrição
TInt8	TUint8		Inteiros de 8 bits com e sem sinal.
TInt16	TUint16		Inteiros de 16 bits com e sem sinal.
TInt32	TUint32		Inteiros de 32 bits com e sem sinal.
TInt	TUint		Inteiros com e sem sinal. Na prática, significa o mesmo que TInt32 e TUint32.
TReal32	TReal64	TReal	Números de ponto flutuante de precisão simples e dupla de acordo com a norma IEEE 754. Equivalentes ao float e ao double. TReal equivale a TReal64.
Tbool			Boolean.
TAny			Equivalente ao ponteiro void, e usado normalmente como TAny* (um ponteiro para qualquer coisa).

Além disso, o Symbian OS define algumas formas diferentes de compilação dos códigos-fonte. Estes são ligeiramente diferentes dos encontrados no C++ para aplicações *desktop*. Aqueles mais usados são listados abaixo:

- EXE (.exe): um programa com um único ponto de entrada E32Main(). A interface com o usuário é limitada a uma janela de console (algo similar ao DOS). Um detalhe deste tipo de aplicação é que ela não aparece no menu do celular, sendo necessária outra aplicação para iniciá-la.
- *Dinamic Link Library* (.dll): uma biblioteca de códigos com vários pontos de entrada. DLL's são carregadas por outros programas.
- *Application* (.app): programas com uma interface gráfica com o usuário. Esta aplicação instala um ícone no menu do celular, podendo ser iniciada diretamente.

Um projeto de desenvolvimento em Symbian é composto por vários arquivos, além dos arquivos com o código fonte. Os principais arquivos são *bld.inf* e .mmp. Resumidamente, eles são usados para direcionar a compilação, de forma similar ao *makefile* encontrado nos sistemas operacionais UNIX. Existem outros arquivos que são muito importantes para o desenvolvimento de interfaces gráficas. Estes não são essenciais neste momento e serão explicados no capítulo 4.

Para exemplificar e ilustrar estes e outros conceitos será construído um programa simples – o *HelloWorld* – em aplicação do tipo EXE. O nome do programa é "hellotext", e é composto de três arquivos: bld.inf, hellotext.mmp e hellotext.cpp e sua única função é escrever "Hello Text" na tela. Este é o primeiro passo no aprendizado do desenvolvimento de aplicações para Symbian OS.

3.3 DEFINIÇÃO DE COMPONENTES

O *bld.inf* é um descritor de componentes. Basicamente, ele indica para o ambiente de desenvolvimento como criar o script de compilação (por exemplo, quais especificações de projeto devem ser usadas para criar os arquivos na construção da aplicação desejada). Este arquivo é apresentado na Listagem 3.1, onde a linha 1 indica que ele irá fazer as definições de componentes e a linha 2 indica em qual arquivo o compilador deverá procurar essas definições. Este é um arquivo de texto comum, que pode ser construído com qualquer editor de texto não formatado (como, bloco de notas, VI ou *UltraEdit*).

Listagem 3.1 Conteúdo do arquivo bld.inf.

```
   // BLD.INF
1. PRJ_MMPFILES
2. Hellotext.mmp
```

3.4 ESPECIFICAÇÕES DE PROJETO

O arquivo de especificações de projeto (.mmp) define as propriedades dos componentes de projeto de uma forma independente da plataforma e do compilador. Ele é usado, junto com o bld.inf, pelo ambiente de desenvolvimento para criar o script de compilação da sua aplicação. A Tabela 3-2 apresenta os campos mais importantes do arquivo e suas funções.

Tabela 3-2 Principais tipos básicos no Symbian C++

Campos	Função
TARGET	O nome da aplicação com a extensão (.exe,.app,.dll).
TARGETTYPE	Especifica o tipo de aplicação. Existem 16 tipos, mas os mais usados são dll, app e exe. Esse parâmetro define o valor do UID1 (descrito na próxima seção).
UID	Determina o valor da UID2 e UID3.
TARGETPATH	Localização da aplicação e seus componentes no dispositivo.
LANG	Suporte a idiomas. Cada idioma tem um código de dois dígitos.
SOURCEPATH	Localização dos arquivos fonte.
SOURCE	Referência aos arquivos fonte.
RESOURCE	Referência aos arquivos de recursos (.rss).
USERINCLUDE, SYSTEMINCLUDE	Localização dos arquivos de *header*.
LIBRARY	Lista de bibliotecas necessárias para a aplicação.

Listagem 3.2 Conteúdo do arquivo hellotext.mmp

```
1.  // hellotext.mmp
2.  TARGET HelloText.exe
3.  TARGETTYPE exe
4.  UID 0
5.  SOURCEPATH .
6.  // O código fonte
7.  SOURCE hellotext.cpp
8.  USERINCLUDE .
9.  SYSTEMINCLUDE \epoc32\include
10. // As bibliotecas
11. LIBRARY euser.lib
```

O conteúdo deste arquivo no projeto de helloworld é mostrado na Listagem 3.2. A linha 2 define que o nome do executável, do exemplo, será HelloText.exe, e a linha 3 define que é uma aplicação do tipo EXE. A explicação para a linha 4 será vista na próxima seção. A linha 5 define que o diretório atual contém os códigos-fonte, assim como a linha 8 define que os cabeçalhos definidos pelo usuário também se encontram no diretório atual. A linha 7 define especificamente quais arquivos formam o código-fonte do nosso exemplo. Por fim, a linha 9 e a linha 11 incluem os cabeçalhos e as bibliotecas do sistema necessárias para o exemplo.

3.5 UID – UNIQUE IDENTIFIER NUMBER

Um UID (*Unique Identifier Number*) é um identificador global de 32 bits. Esse número funciona da mesma forma que a extensão de um arquivo do Windows, que é usada para

definir o tipo do arquivo. Além disso, todo arquivo criado por uma aplicação tem o UID da aplicação associada ao arquivo, permitindo que o sistema operacional relacione o arquivo à aplicação que o gerou, de uma forma similar ao Windows associa a extensão ".doc" ao Word.

O UID possui três componentes: UID1, UID2 e UID3. O UID1 é definido pelo tipo de aplicação do campo TARGETTYPE do arquivo .mmp do projeto. O UID2 define um subgrupo para tipos de arquivos que compartilham o mesmo UID1. O UID3 serve para distinguir cada aplicação, e este é usado pelo sistema operacional para associar documentos a aplicativos. O desenvolvedor deve pedir para a própria Symbian o seu UID3, de modo que dois aplicativos diferentes não compartilhem o mesmo UID3. Para isso, deve-se enviar um e-mail para uid@symbiandevnet.com com o assunto '*UID request*' e sua UID será recebida por e-mail. Na fase de testes, entretanto, podem ser usados alguns dentre estes UID3 que a Symbian alocou para este propósito: 0x01000000 – 0x0fffffff.

Para o projeto EXE, os campos UID2 e UID3 não são necessários, porém utilizar 0 como UID2 evita a geração de um aviso pelo compilador (conforme utilizado na linha 4 da Listagem 3.2).

3.6 CONSTRUINDO O CÓDIGO

As aplicações Symbian são nativas para os dispositivos aos quais se destinam. Isso significa que ela lida diretamente com o hardware do aparelho e que, portanto, a construção final do executável deve ser compatível com o *assembly* do processador do aparelho. Isso é importante pois diferentemente de uma aplicação Java, o emulador não consegue rodar o arquivo construído para o aparelho celular, uma vez que a arquitetura do processador do computador é compatível com a arquitetura x86 (Pentium, AMD), o que não ocorre com os processadores Texas ARM encontrados na maior parte dos aparelhos Symbian.

Por essa razão deve-se ter cuidado ao compilar e construir uma aplicação Symbian. Se a plataforma alvo desejada for o emulador de PC, então a compilação deve ser feita para o tipo WINS (*WINdows Single Process*) e o comando fica "abld build winsb udeb". Se a plataforma-alvo desejada for o dispositivo real, a compilação deve ser feita para o tipo ARMI e o comando fica "abld build armi udeb". Durante o desenvolvimento usam-se os dois tipos – um para testar a aplicação no PC e outro para testar no dispositivo real. É

importante notar que o arquivo compilado e construído para um dos tipos não encontrará nenhum problema para ser instalado na outra plataforma, mas não conseguirá ser executado.

Além da plataforma-alvo, existem mais dois tipos de construções possíveis: a compilação do tipo "*debug*" (udeb) e a compilação do tipo "*release*" (urel). A compilação para debug habilita uma série de funcionalidades, como algumas asserções (quando utilizadas com a macro __ASSERT_DEBUG) e a verificação de balanço de *heap* (realizadas pelas macros __UHEAP_MARK e __UHEAP_MARKEND) e que serão explicadas nos item 3.9. Essas macros são desabilitadas quando a construção é feita no tipo "release", pois a aplicação estável e comercial não precisaria mais delas. O comando para construir no tipo "debug" é "abld build armi udeb" e o comando para construir no modo "release" é "abld build armi urel".

3.7 EXPLICANDO O CÓDIGO

O código utilizado no exemplo pode ser visto na Listagem 3.3. Existem ainda outros arquivos que precisam ser incluídos no projeto e serão mostrados nos outros itens desse capítulo.

Listagem 3.3 Código fonte do programa – hellotext.cpp.

```
// hellotext.cpp
1. #include <e32base.h>
2. #include <e32cons.h>
3. LOCAL_D CConsoleBase* gConsole;
// Função Main - Aqui se encontra a lógica do programa.
4. void MainL()
5. {
6. gConsole->Printf(_LIT("Hello world!\n"));
7. }
// Controle do Console
8. void ConsoleMainL()
9. {
// Cria um console
10. gConsole = Console::NewL(_LIT("Hello Text"),
11. TSize(KConsFullScreen, KConsFullScreen));
12. CleanupStack::PushL(gConsole);
// Chama a função Main
13. MainL();
// Pausa a aplicação
14. User::After(5000000); // 5 second delay
// Tira o console da Cleanup Stack e destrói
15. CleanupStack::PopAndDestroy(gConsole);
16. }
// Controle da Cleanup stack
17. GLDEF_C TInt E32Main()
18. {
19. __UHEAP_MARK;
20. CTrapCleanup* cleanupStack = CTrapCleanup::New();
21. TRAPD(error, ConsoleMainL());
22. __ASSERT_ALWAYS(!error, User::Panic(_L("SCMP"), error));
23. delete cleanupStack;
24. __UHEAP_MARKEND;
25. return 0;
26. }
```

Pode parecer estranho que o código do HelloWorld – a mais simples das aplicações – precise de três funções só para escrever uma simples seqüência de caracteres em um console. A resposta para isso é que o sistema Symbian não é otimizado para realizar programas mínimos – ao contrário, é otimizado para as aplicações reais. Além disso, o programa da Listagem 3.3 gerencia todos os erros que podem ocorrer.

As funções usadas são:

- MainL() : faz o trabalho de escrever "*Hello World!*" na tela.
- ConsoleMainL(): aloca um console e chama MainL().
- E32Main(): ponto de entrada de uma aplicação em console. Chama ConsoleMainL() dentro de função de controle de erro (*trap harness*) .

Alguns conceitos serão explicados de forma sucinta aqui pois serão retomados com mais detalhes ao longo do texto, assim como o estilo de codificação. A função MainL() faz uma chamada ao método printf, como esperado. Programadores em C irão reconhecer este método, porém poderão estranhar sua chamada. Ele aparece como "gConsole->Printf()" (Listagem 3.3, linha 6) porque o Symbian C++ é orientado a objetos – a função Printf() é membro da classe CConsoleBase. A macro _LIT(), que é passada como argumento da função Printf(), transforma uma string comum em C para um descritor (*descriptor*), que é a forma como o Symbian C++ trata *strings*.

A função E32Main() (Listagem 3.3, linha 17) é o ponto de entrada da aplicação. Juntamente com a função ConsoleMainL (Listagem 3.3, linha 8), ela constrói os dois componentes de infra-estrutura que a função MainL() (Listagem 3.3, linha 13) precisa para realizar o seu trabalho – uma *cleanup stack* (pilha de limpeza) e um console. A declaração de E32Main() indica que ela é uma função global (GLDEF_C). O tipo de retorno da função é TInt, ao invés de simplesmente "int". Um "int" poderia ser usado, porém não é padrão em programas Symbian C++. O importante agora sobre essa função é o gerenciamento de erros.

A macro TRAPD() (Listagem 3.1Listagem 3.3, linha 21) funciona como o try-catch do Java ou C++ - ela captura qualquer função que falha (em inglês, *leave*). Isso é o que significa o L no final do nome das funções MainL() e ConsoleMainL() – que elas podem falhar (*leave*), e que elas não gerenciam essa falha internamente (similar à cláusula throws em Java). Essa convenção de colocar um L no final é muito importante.

A função ConsoleMainL() aloca um console antes de chamar MainL() para mostrar a mensagem. Após isso, ela pausa por cinco segundos antes de finalizar a execução. Não é necessário tentar gerenciar o erro na chamada a MainL() – caso ele aconteça, a função ConsoleMainL() também irá falhar e esta falha será capturada pela armadilha (*trap*) armada na função E32Main().

3.8 COMPILAR E CONSTRUIR O CÓDIGO EXECUTÁVEL E ARQUIVO DE INSTALAÇÃO

Para compilar e gerar um executável serão necessários os três arquivos mostrados itens anteriores desse capitulo, “bld.inf”, “hellotext.mmp” e “hellotext.cpp”, situados no mesmo diretório. Outro arquivo que deve estar nesse diretório, “HelloText.pkg”, é explicado mais tarde. Além disso, o Series 60 SDK 1.2 for Symbian OS, mostrado no Apêndice C ,foi considerado instalado no diretório $RAIZ:\Symbian\Series60_1_2_B. Esse item mostra como criar o arquivo para rodar no celular via linha de comando para entender o que as IDE, mostradas no Apêndice C fazem.

Os passos para criação do executável em linha de texto são:

- Algumas variáveis de ambiente deverão estar configuradas. Após instalar Borland C++ Mobile Edition for Series 60, configurar as seguintes variáveis em uma janela de linha de comando:

```
set EPOCROOT=\Symbian\Series60_1_2_B\
set PATH=C:\Symbian\Series60_1_2_B\epoc32\tools
```

Esses valores são válidos somente para a janela DOS na qual os comandos foram chamados. Se desejável, configurar as variáveis de ambiente do computador de forma definitiva.

- Usando a mesma janela onde foram executados os comandos de configuração de ambiente, entrar no diretório onde se encontram os arquivos mostrados anteriormente.
- Digitar o comando abaixo:

```
Bldmake bldfiles
```

Este comando utiliza os arquivos bld.inf e o hellotext.mmp para gerar o arquivo abld.bat, que contém as diretrizes para a compilação. O abld.bat é gerado automaticamente pelo comando bldmake e não deve ser modificado manualmente.

- Digitar o comando abaixo:

```
Abld build winsb udeb
```

Esse comando vai construir o executável para o emulador do terminal Serie 60. A explicação da sintaxe desse comando foi apresentada no item 3.6. O arquivo executável é criado no local:

```
C:\Symbian\Series60_1_2_B\epoc32\release\winsb\udeb\HELLOTEXT.EXE
```

O resultado da execução no programa no computador é mostrado na Figura 3.1

Figura 3.1 Simulação da aplicação “*HelloText*”.

- Para executar no dispositivo real é necessário recriar o executável para o processador ARM com o comando:

```
Abld build armib urel
```

Após esse comando, cria-se o arquivo:

```
C:\Symbian\Series60_1_2_B\epoc32\release\armi\urel\Hellotext.exe
```

Esse arquivo não rodará no Windows, sendo possível executá-lo somente no dispositivo real.

- O próximo passo é criar um arquivo de instalação para o celular denominado .sis. O comando para isso é:

```
makesis HelloText.pkg
```

O arquivo HelloText.pkg contém as diretivas para criar o pacote de instalação .sis. O resultado é o arquivo HelloText.sis, que é criado no mesmo diretório onde está o código-fonte, e que pode ser transferido para o celular e instalado. Por ser uma aplicação console, ela não instala um ícone na área de trabalho do telefone, e deve ser usado um outro programa para executar o arquivo “hellotext.exe”. Uma possibilidade é o FExplorer, um programa que funciona como o *explorer* do Windows. Utilizando este programa, basta encontrar o arquivo no local onde ele foi instalado e selecionar “abrir”.

3.9 VERIFICAÇÃO DE BALANÇO DA HEAP

Uma ferramenta que auxilia os programadores a liberar os recursos alocados é a checagem de balanço da *heap* (*heap balance checking*). Este é o ponto onde se explicam as macros __UHEAP_MARK e __UHEAP_MARKEND. Elas podem ser utilizadas em qualquer parte do seu programa, inclusive criando alinhamentos.

Cada objeto criado com a função new() (ou uma de suas variantes) deve ser destruído com um comando delete. A macro __UHEAP_MARKEND verifica se a *heap* está balanceada. Se a *heap* não tiver o mesmo número de células alocadas em relação ao que tinha quando __UHEAP_MARK foi chamado, a aplicação entra em pânico.

Esta funcionalidade assegura que o desbalanceamento da *heap* seja detectado ainda no desenvolvimento e está disponível em todas compilações de *debug* (debug *builds* - udeb) do Symbian OS (inclusive no emulador).

Na Listagem 3.3 essas macros são utilizadas nas linhas 19 e 24, respectivamente, e é exatamente o primeiro ponto de entrada da aplicação e a última linha executada antes da aplicação retornar, assegurando que a aplicação retornou ao sistema operacional sem deixar lixo na memória.

3.10 GERENCIAMENTO DE ERROS

A chamada à função User::Leave() faz a execução da função atual terminar, e de todas as outras funções que a chamaram, até encontrar uma macro TRAP() ou TRAPD(). A função User::Leave() recebe um inteiro de 32 bits como parâmetro, embora normalmente sejam usados códigos já definidos, como o código KErrNone, que é definido para indicar que não houve erros. Estes códigos servem para indicar qual a causa do erro (se ele foi um erro de memória, de i/o, etc.) à função que chamou.

O código das macros TRAP() e TRAPD() é bastante confuso. O importante é compreender que a macro TRAP() chama uma função (o seu segundo parâmetro) e retorna o código de falha (o seu primeiro parâmetro), caso ocorra a falha. Se a falha não ocorrer, ela retorna ao código KErrNone (que é definido como zero). A única diferença em relação à macro TRAPD() é que esta declara a variável de erro, primeiro, salvando uma linha de código.

A forma de utilizar o TRAP pode ser vista na Listagem 3.4.

Listagem 3.4 - Utilizando Traps.

```
1. TRAPD(r, pR->RunL());
2. If (r != KErrNone)
3.     os->Error(r);
```

A macro TRAPD() na linha 1 faz uma chamada à função RunL() da classe pR. Pela nomenclatura da função, sabe-se que ela pode falhar – e exatamente por isso ela é chamada envolta a uma macro TRAP. Além disso, a macro TRAPD() declara a variável r como sendo um TInt. Se fosse utilizada a macro TRAP, isso deveria ser feito antes. Após o retorno da função RunL() (com ou sem erro) o programa segue sua ordem, sendo executada a linha 2. Se a função terminar por conseqüência de uma falha, retorna ao código dessa falha na variável r. Se terminou normalmente, é retornado o código

KErrNone. Assim, a linha 2 checa o valor da variável r e, se houver algum erro, lida com ele através da chamada à função Error do objeto os (linha 3).

3.11 FUNÇÕES L

É muito importante saber se uma determinada função pode falhar e isso é indicado por um L no final do nome da função (como em RunL()). Entretanto, duas considerações devem ser feitas neste ponto:

- Se uma dada função chamar uma outra função que pode falhar, então esta função também deve avisar que pode falhar (com o L no final) ou gerenciar a falha internamente (através das macros TRAP() e TRAPD()).
- Se uma dada função chama new (ELeave) ela também deve ser uma função que pode falhar.

Por fim, a convenção de nome L é muito importante, pois o compilador não checa se uma função pode falhar. O código compila normalmente com ou sem o L ao final, e o problema só é observado em tempo de execução.

3.12 CLEANUP STACK (PILHA DE LIMPEZA)

A *cleanup stack* surgiu para resolver o problema de limpeza de objetos alocados na *heap*, cujo único ponteiro é uma variável automática. Se a função que tem os objetos alocados falhar, estes objetos precisam ser liberados. Considerando que o Symbian OS não utiliza as exceções do C++ é necessário criar um novo mecanismo para garantir que isso ocorra – a *cleanup stack*. A Listagem 3.5 apresenta um trecho de código que ilustra esse problema.

Listagem 3.5 - Utilizando a *Cleanup Stack* – parte 1.

```
1. Object* x = new (ELeave) Object;
2. x->DoSomethingL();
3. delete x;
```

E a função DoSomethingL:

```
1. void Object::DoSomethingL()
2. {
3. TInt* y = new (ELeave) TInt;
4. delete y;
5. }
```

Este código pode falhar em dois momentos: na primeira alocação do objeto Object, por causa do new (ELeave); ou na função DoSomethingL(), pelo mesmo motivo. Se

a função falhar no primeiro momento, o código interrompe sua execução automaticamente sem nenhum problema. Entretanto, se o código falhar no segundo momento, a função DoSomethingL() é interrompida e o mesmo acontece com código que a chamou, perdendo a referencia do ponteiro x sem nunca chamar a instrução delete x para liberar o recurso. Assim, um trecho da *heap* ficará perdido, a verificação da *heap* falhará e a aplicação entrará em pânico.

A *cleanup stack* é utilizada para manter um ponteiro para um objeto. Objetos na *cleanup stack* são automaticamente destruídos se uma falha ocorrer. A Listagem 3.6 apresenta o código para o exemplo acima utilizando a *cleanup stack*.

Listagem 3.6 Utilizando a *Cleanup Stack* – parte 2.

```
1. Object* x = new (ELeave) Object;
2. CleanupStack::PushL(x);
3. x->DoSomethingL();
4. CleanupStack::PopAndDestroy(x);
```

O que ocorre agora com o código é o seguinte:

- Imediatamente após alocar o objeto Object e colocar seu ponteiro em x (linha 1), este ponteiro é colocado na *cleanup stack* pelo método PushL() (Listagem 3.6,linha 2);
- Então é feita a chamada a DoSomethingL() (Listagem 3.6**,**linha 3);
- Caso não ocorra a falha, o ponteiro é retirado da *cleanup stack* e destruído (Listagem 3.6 linha 4). Como esta é uma operação comum, já existe a função que faz as duas coisas em uma única chamada (CleanupStack::PopAndDestroy());
- Caso a falha ocorra, o código falha normalmente. Entretanto, dessa vez todos os objetos com ponteiros na *cleanup stack* são destruídos como parte do processo de falha.

3.13 CONSTRUTORES DE SEGUNDA-FASE

A *cleanup stack* é usada para manter ponteiros para objetos na *heap* de forma que eles possam ser liberados se uma falha ocorrer. Se uma falha acontecer na construção de um objeto a referência para ele é perdida, mesmo que você tenha usado a *cleanup stack*, fazendo a aplicação entrar em pânico.

Isso significa que deve haver a chance de colocar o ponteiro para este objeto na *cleanup stack*. Caso uma falha ocorra na alocação, esta não poderá ser realizada. Isto forçou a uma regra do Symbian C++: os construtores não podem falhar. Mas o construtor é um

lugar em que geralmente se aloca a memória. Paradoxalmente, isso é uma ação que tem um grande potencial para falhar. Para corrigir este impasse, foram instituídos os construtores de segunda fase.

No Symbian C++, é convenção fazer a construção de um objeto por partes, criando uma nova função ConstructL(), que pode falhar, atuando como um construtor de segunda fase. A Listagem 3.7 descreve como seria o código.

Listagem 3.7 - Utilizando Construtores de segunda fase – parte 1.

```
1. Object* x = new (ELeave) Object;
2. CleanupStack::PushL(x);
3. x->ConstructL(); //construtor de segunda fase.
4. x->DoSomethingL();
5. CleanupStack::PopAndDestroy(x);
```

Como pode ser visto no código, quando se quer criar um objeto qualquer "x" deve-se chamar o construtor dele (Listagem 3.7**,** linha 1), que deverá apenas criar a referência para o objeto, coloca na *cleanup stack* (Listagem 3.7**,**linha 2), e então chama o construtor de segunda fase (Listagem 3.7**,** linha 3), que deverá iniciar as variáveis do objeto "x". Assim, o código pode prosseguir normalmente na linha 4, chamando uma função qualquer do objeto, e destruindo o objeto na linha 5, quando não for mais necessário.

Como visto, são necessárias 3 linhas para inicializar e usar um objeto, devido ao construtor de segunda fase e *cleanup stack*. Por essa razão, é também comum encontrar outra versão do operador new(), chamada newL(). A função newL() já cria o seu objeto completamente, lidando com o problema da *cleanup stack* e do construtor de segunda fase. O código para isso deve ser feito pelo programador, e normalmente é feito como na Listagem 3.8. A Listagem 3.9 mostra como seria a Listagem 3.7 se a função newL() fosse utilizada.

Listagem 3.8 - Utilizando Construtores de segunda fase – parte 2 – newL().

```
1. Object* Object::NewL()
2. {
3. Object* self = new(ELeave) Object;
4. CleanupStack::PushL(self);
5. Self->ConstructL();
6. CleanupStack::Pop(self);
7. Return self;
8. }
```

Listagem 3.9 - Utilizando o operador newL().

```
1. Object* x = newL (ELeave) Object;
2. CleanupStack::PushL(x);
3. x->DoSomethingL();
4. CleanupStack::Pop(x);
```

A função NewL() é declarada estática. Assim, ela pode ser utilizada no lugar de apenas new(), retornando o ponteiro do objeto construído adequadamente (se nenhuma falha ocorrer). Note que a função que está sendo chamada, doSomethingL(), pode falhar (o "L" no final do nome indica isso), o objeto "x" deve ser colocado na *Cleanup Stack*, pois aponta para a *heap,* cuja única referência é uma variável automática. Por isso, outra função comumente encontrada é a newLC(), que é muito semelhante à função newL(), tendo a única diferença de não retirar o ponteiro para este objeto da *cleanup stack* (e o programador deve fazer isso mais tarde). Um exemplo de código pode ser visto na Listagem 3.10 , assim como o seu uso pode ser visto na Listagem 3.11.

Listagem 3.10 - Utilizando Construtores de segunda fase – parte 2 – newLC().

```
1. Object* Object::NewLC()
2. {
3. Object* self = new(ELeave) Object;
4. CleanupStack::PushL(self);
5. Self->ConstructL();
6. Return self;
7. }
```

Listagem 3.11 - Utilizando o operador newLC().

```
1. Object* x = newLC (ELeave) Object;
2. x->DoSomethingL();
3. CleanupStack::Pop(x);
```

É importante notar que essas funções são convenções e opcionais – elas podem ou não estar definidas nas suas classes ou em classes que você estiver trabalhando.

3.14 CONCLUSÃO DO CAPITULO

Neste capítulo, a história e a introdução do Symbian OS e da linguagem de desenvolvimento Symbian C++ foram mostradas. O próximo capitulo mostra uma introdução a interface gráfica e que existem três formas de criá-la.

4 INTERFACES GRAFICAS EM SYMBIAN C++

Interfaces gráficas facilitam o uso de um software pelo usuário. Apresentar o conteúdo de forma visual torna mais agradável ao usuário, permitindo que ele associe funções e comandos a elementos na tela.

No capítulo anterior foram vistos os fundamentos do Symbian OS, bem como as principais diferenças de sua linguagem de programação nativa, o Symbian C++, para o C++ para *desktops*. Neste capítulo, demonstraremos uma aplicação com interface gráfica.

Para o desenvolvimento de aplicações com interface gráfica no Symbian, funcionalidades, que são agrupadas em um único arquivo ou classe em outras plataformas de desenvolvimento, são separadas desde a concepção. Isso facilita na tarefa de separar a lógica de um programa de sua parte visual. No entanto, infelizmente, isso torna mais difícil o entendimento de uma aplicação gráfica em Symbian.

4.1 ARQUITETURA DE UMA APLICAÇÃO

As aplicações com interface gráfica em Symbian devem apresentar as seguintes funcionalidades:

- Exibir graficamente informações para o usuário e permitir interatividade;
- Responder a eventos iniciados pelo usuário;
- Responder a eventos iniciados pelo sistema;
- Ter a capacidade de salvar e recuperar os dados da aplicação.
- Identificar-se unicamente para o sistema operacional.
- Exibir informações sobre a aplicação para o sistema operacional.

Existe um núcleo básico de classes que formam a aplicação, e algumas classes do sistema operacional que este núcleo deve herdar de forma a obter as funcionalidades básicas para o funcionamento da aplicação.

As classes que compõem a arquitetura da aplicação que provêem estas funcionalidades são divididas em quatro categorias: Application, Document, AppUI e View.

A classe Application tem uma função fixa e bem definida. Ela serve somente como ponto de entrada da aplicação, e disponibiliza informações sobre a aplicação para o sistema operacional, representados pelo ícone e nome da aplicação, que devem aparecer

no gerenciador de aplicativos, e o UID (*Unique Identifier Number* – o número pelo qual as aplicações Symbian são identificadas) da aplicação. Esta classe tem um papel estático – ela não se envolve com os dados da aplicação ou com a sua lógica.

A classe de Document provê um contexto para a persistência dos dados da aplicação. Esta classe contém métodos para instanciar a classe de AppUI.

A classe de AppUI é um recipiente para várias notificações vindas do sistema operacional, como eventos de teclado ou de sistema. Esta classe irá gerenciar a forma como a aplicação irá tratar esses eventos, despachando para o View, que deveria receber a notificação desse evento (por exemplo, a tela de formulário é a tela que deveria ser avisada de que o usuário finalizou o preenchimento desse formulário).

Um *View* é um conceito que significa "a representação dos dados da aplicação na tela". O *View* não é uma categoria de classe de forma estrita. Existem essencialmente três formas diferentes de se fazer o *View* no Série 60, como será mostrado na seção 6.5.

4.2 O FRAMEWORK DO SISTEMA OPERACIONAL

O *Framework* do Sistema Operacional é a maneira pela qual uma aplicação interage com o sistema operacional. Por exemplo, quando o sistema operacional deseja finalizar uma aplicação, ele se comunica com a aplicação por meio deste *framework*. Ele é formado por um conjunto de classes núcleo (*core classes*) que formam a base para todas as aplicações. Estas classes formam a estrutura necessária, e também encapsulam a comunicação entre a aplicação e o sistema operacional.

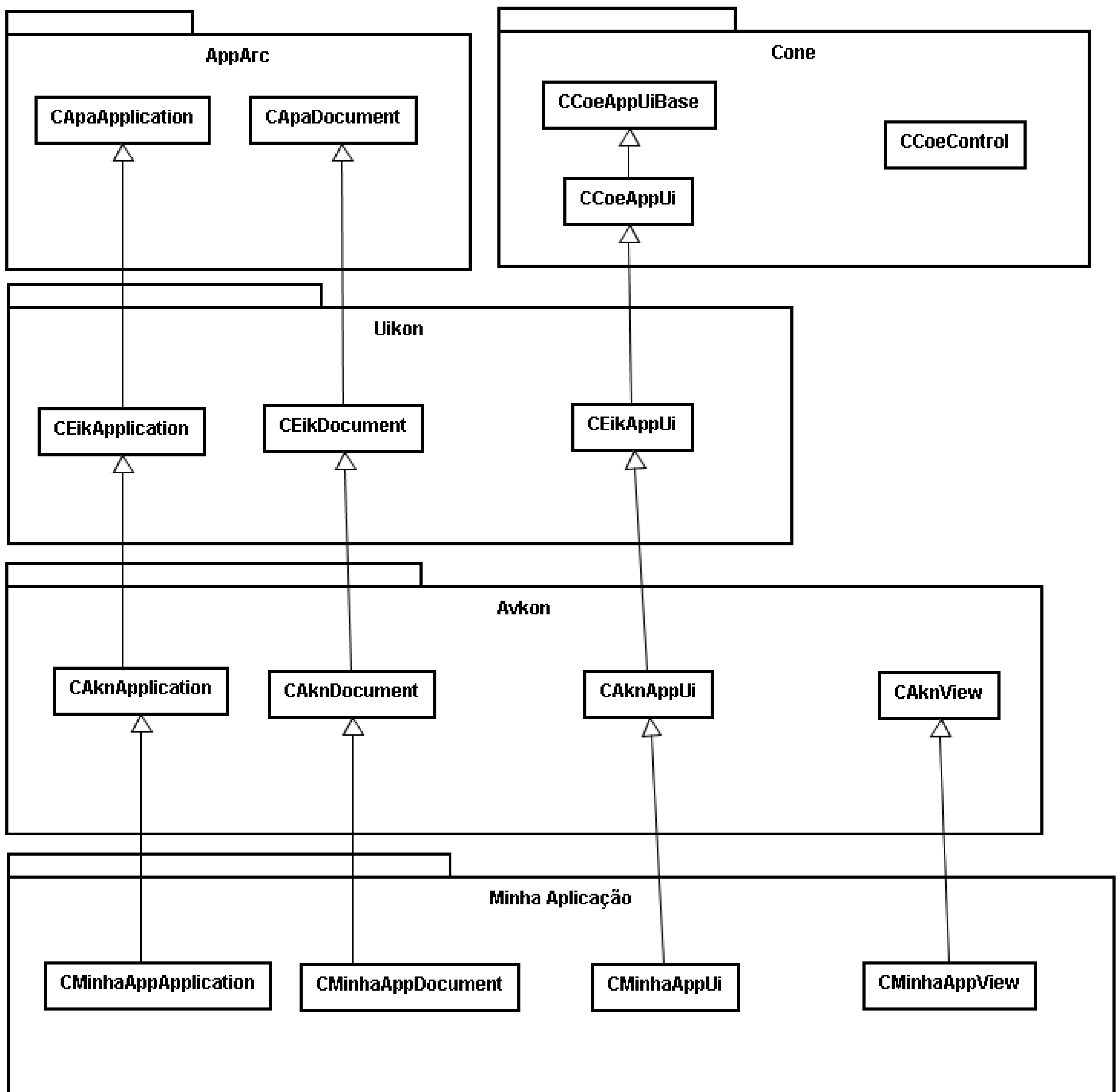

Figura 4.1 - Diagramas das classes núcleo.

A Figura 4.1 apresenta um diagrama UML mostrando quatro camadas de classes, simplificado para facilitar o entendimento.

A primeira camada é dividida em dois componentes fundamentais – o AppArc (*Application Architecture* – arquitetura da aplicação) e o CONE (*CONtrol Enviroment* – ambiente de controle).

As classes no AppArc provêem a estrutura básica da aplicação e os mecanismos para retornar ao sistema as informações sobre a aplicação e sobre os dados persistentes. Este é um componente do Symbian OS e suas classes têm o nome iniciado com o prefixo "*Apa", como em CApaApplication.

As classes no CONE provêem os mecanismos básicos para gerenciar a entrada do usuário e criar a interface gráfica. Este também é um componente do Symbian OS e suas classes têm o nome iniciado por “*Coe”, como em CCoeControl.

A segunda camada de classes pertence ao componente Uikon, também chamado Eikon. Este componente contém as implementações de interfaces gráficas que são independentes de plataforma dos diversos recursos do Symbian OS. As classes desse componente têm o nome iniciado por “*Eik”, como em CEikApplication.

A terceira camada de classes é uma implementação das interfaces gráficas específica do Uikon para o Série 60, e é chamada Avkon. A escolha de qual implementação (Avkon ou Uikon) será explicada ao mais adiante no capitulo, na seção 4.4.As classes pertencentes a este componente têm o nome iniciado por “*Akn”, como em CAknApplication.

A quarta camada é específica da aplicação, e demonstra como podem ser derivadas as classes de modo a construir sua própria aplicação.

A maior parte das classes na primeira camada é abstrata, definindo apenas as interfaces, com o *framework*, que serão utilizadas. A segunda camada adiciona implementações comuns ao Symbian OS, enquanto a terceira adiciona implementações específicas para o Série 60. Por fim, a quarta adiciona as implementações específicas da sua aplicação.

As classes que compõem a infra-estrutura da aplicação – Application, Document, AppUI e AppView – serão mais detalhadas nas seções a seguir.

4.2.1 Classe Application

A classe da aplicação deve derivar de CEikApplication (ou de uma outra classe que deriva desta, por exemplo, CAknApplication – ver Figura 4.1). Existem duas funções que devem ser implementadas por esta classe. A primeira função, CreateDocumentL(), é utilizada para criar um documento e, por isso, deverá retornar um objeto do tipo CApaDocument. A segunda função, AppDllUID(), deverá identificar a aplicação, retornando seu UID.

Uma implementação comum dessas funções aparece na Listagem 4.1

Listagem 4.1 Implementação comum da classe Application.

```
//Este número deve ser o mesmo encontrado no campo UID3 do arquivo .mmp
static const TUid KUidMinhaApp = {0x10005B91};

//Cria um documento da aplicação, e retorna um ponteiro para ele.
CApaDocument* CMinhaAppApplication::CreateDocumentL() {
   CApaDocument* document = CMinhaAppDocument::NewL(*this);
   return document;
}
```

```
//Retorna o UID da aplicação.
TUid CMinhaAppApplication::AppDllUid() const {
   return KUidMinhaApp;
}
```

Na linha 2, é definido uma constante estática KUidMinhaApp, que deve conter estritamente o mesmo valor encontrado no arquivo .mmp da aplicação (de especificações de projeto). Esse valor é retornado pela função AppDllUid(), identificando a aplicação para o sistema operacional. Essa identificação é importante para que o sistema operacional possa relacionar arquivos à sua aplicação. A outra função definida, CreateDocumentL(), é utilizada para criar um documento-padrão.

A grande maioria das aplicações Symbian, que utiliza GUI, implementa essas funções de forma semelhante acima exposto. Este é o comportamento esperado pelo sistema operacional de *qualquer* aplicação Symbian.

4.2.2 Classe Document

A classe do documento deve derivar de CEikDocument (ou de uma classe que deriva desta, como por exemplo, CAknDocument – ver Figura 4.1). Em geral, esta classe instancia a lógica da aplicação e cria um objeto do tipo AppUI por uma chamada ao método CreateAppUiL(). O AppUI é necessário para manipular os eventos de sistema.

Neste momento, o código começa a se tornar mais específico para cada aplicação. Os códigos vistos nas Listagem 4.1 e Listagem 4.2 são obrigatórios e possuem uma função muito bem-definida, sendo pouco flexíveis. Eles devem obedecer a um comportamento específico, não sendo permitido desvios deste comportamento.

Em uma aplicação que gerencia e modifica dados persistentes, a classe Document é a classe que essencialmente representa a persistência no programa, sendo responsável por conter os métodos para criar, abrir e modificar os arquivos de dados.

Um trecho do código desta classe pode ser visto na Listagem 4.2. Este código apenas cria um objeto da classe AppUI e retorna um ponteiro para ele.

Listagem 4.2 Implementação comum da classe Document.

```
// Cria um objeto AppUI e retorna um ponteiro para ele.
CEikAppUi* CMinhaAppDocument::CreateAppUiL() {
   CEikAppUi* appUi = new (ELeave) CMinhaAppUi;
   return appUi;
}
```

4.2.3 Classe AppUI

A classe AppUI deve derivar de CEikAppUI (ou de alguma classe que deriva desta, como CAknAppUI – ver Figura 4.1) que, por sua vez, deriva de CCoeAppUI, pertencente ao Controle do Ambiente (*Control Enviroment* – CONE). É neste ponto que a ação de GUI se inicia efetivamente e esta classe apresenta duas tarefas principais:

- Capturar os comandos para a aplicação.
- Distribuir os eventos de teclas para os vários *views* da aplicação.

A maior parte das funções dessa classe não é puramente abstrata ("virtual" pela linguagem C++, porém o termo "abstrata" é mais esclarecedor), tendo apenas suas implementações vazias. Isso significa que o programador não precisa implementar *todas* essas funções – ele apenas precisa implementar aquelas que ele vai utilizar efetivamente. Algumas das funções importantes implementas nesta classe são brevemente discutidas abaixo:

- HandleKeyEvent() – chamada quando ocorre um evento de tecla pressionada.
- HandleForegroundEventL() – chamado quando a aplicação perde o foco (vai para o *background*) ou obtém o foco novamente.
- HandleCommandL() – chamado quando é selecionado um comando.
- HandleSystemEventL() – chamado quando é gerado um evento de sistema.
- HandleApplicationSpecificEventL() – notificação de eventos customizáveis que se pode definir.

Além disso, esta classe é responsável por criar e gerenciar as *applications views* – que serão descritos a seguir.

4.2.4 Classe AppView

O *application view* deriva do CONE CCoeControl (ou qualquer classe que deriva desta). De acordo com a forma como a arquitetura da aplicação será projetada, uma classe diferente poderá fazer o papel do AppView, mas o AppView sempre estará ligada à classe que efetivamente desenha informações na tela.

4.3 INICIAÇÃO DE UMA APLICAÇÃO

Neste tópico mostraremos os principais passos que o sistema operacional executa desde o momento em que se seleciona uma aplicação para ser executada até o momento em

que esta aplicação será efetivamente executando na tela (o primeiro momento em que ela se desenha na tela). Os passos para isso são apresentados na Figura 4.2.

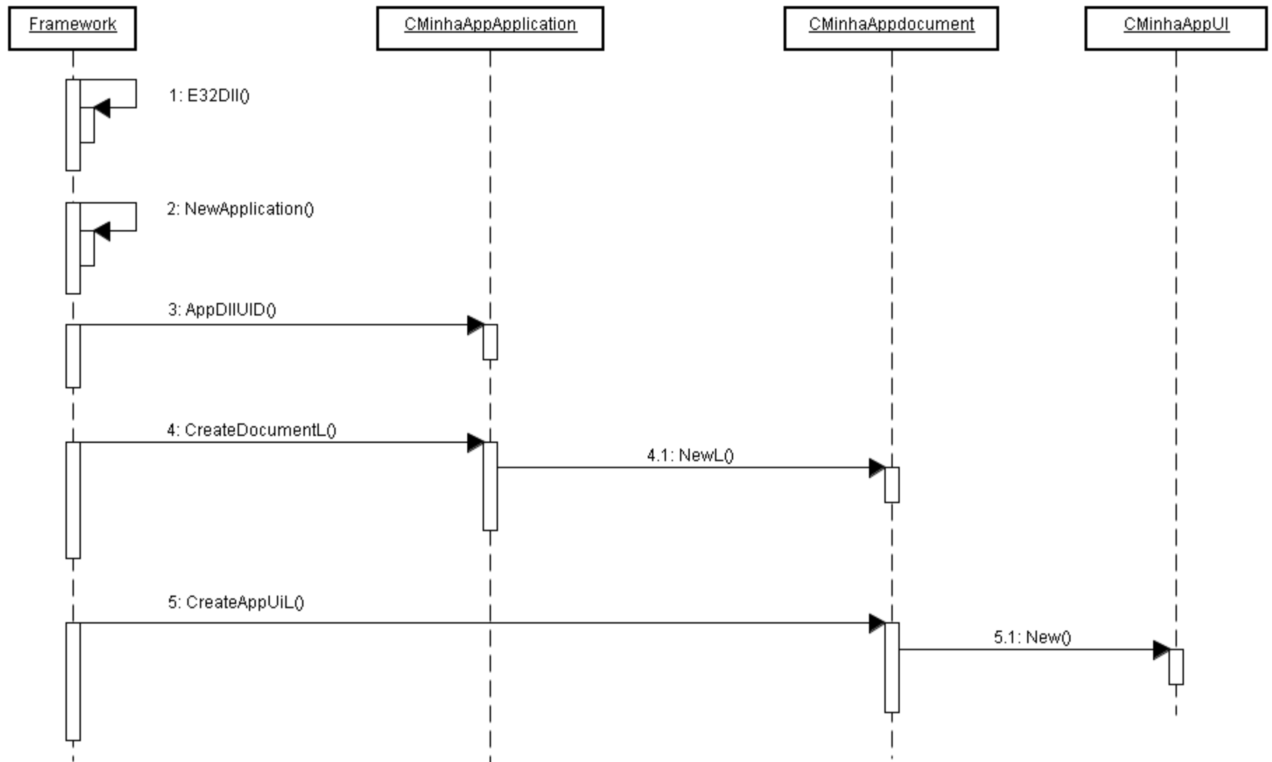

Figura 4.2 Seqüência de inicialização de uma aplicação.

O diagrama da Figura 4.2 pode ser descrito em cinco passos:

- **Passo 1:** Toda aplicação com GUI para série 60 deve implementar a função global E32Dll(). Esta será a primeira função que será chamada pelo sistema operacional, quando ocorrer a tentativa de executar a aplicação. Ela é chamada de "ponto de entrada do DLL" (DLL *entry point*) e deve estar sempre presente na aplicação.
- **Passo 2:** O segundo passo ocorre quando o *framework* do sistema operacional chamar a função NewApplication(). Efetivamente, cria-se o objeto da aplicação, retornando um ponteiro para o sistema operacional. Esta função deve ser exportada pela aplicação. O código dessas duas funções pode ser visto na Listagem 4.3 .

Listagem 4.3-Métodos para inicalização da aplicação.

```
// Ponto de entrada da aplicação.
// Retorna que tudo está correto.
GLDEF_C TInt E32Dll(TDllReason) {
   return KErrNone;
}
```

```
// Cria um objeto da aplicação e retorna um ponteiro para ele.
EXPORT_C CApaApplication* NewApplication() {
   return (new CMinhaApplication);
}
```

- **Passo 3:** Após criar o objeto da aplicação, o *framework* chama a função AppDllUid, encontrada na classe Application da sua aplicação (no nosso exemplo, CMinhaAppApplication), que retorna ao UID3 da aplicação (Listagem 4.4).

Listagem 4.4-Método para identificar a aplicação. Este fragmento encontra-se na classe Application

```
//Retorna o UID da aplicação.
TUid CMinhaAppApplication::AppDllUid() const {
   return KUidMinhaApp;
}
```

O valor retornado deve ser o mesmo encontrado no arquivo .mmp (de especificações de projeto) da aplicação e é usado pelo sistema operacional para determinar se já existe uma instância dessa aplicação em execução.

- **Passo 4:** O próximo passo é a chamada ao método CreateDocumentL() existente na classe *Application* (CMinhaAppApplication, no exemplo). Este método retorna para o *framework* um ponteiro para um objeto do tipo CApaDocument apontando para o objeto da classe Document (CMinhaAppDocument, que estende CAknDocument – ver Figura 4.1). Na linha 3 da Listagem 4.5, um ponteiro para a classe Application é passado para a classe Document, possibilitando chamar o método AppDllUid() para descobrir o UID3 da aplicação e gerenciar os arquivos que são dessa aplicação.

Listagem 4.5 - Método para instanciar a classe Document. Este fragmento encontra-se na classe Application.

```
//Cria um documento da aplicação, e retorna um ponteiro para ele.
CApaDocument* CMinhaAppApplication::CreateDocumentL() {
   CApaDocument* document = CMinhaAppDocument::NewL(*this);
   return document;
}
```

- **Passo 5:** Agora o *framework* usa este ponteiro para a classe Document para criar o objeto AppUI (no nosso caso, CMinhaAppUI), chamando o método CreateAppUiL().

Listagem 4.6 Método para instanciar a classe AppUI. Este fragmento encontra-se na classe Document.

```
// Cria um objeto AppUI e retorna um ponteiro para ele.
```

```
CEikAppUi* CMinhaAppDocument::CreateAppUiL() {
   CEikAppUi* appUi = new (ELeave) CMinhaAppUi;
   return appUi;
}
```

Dessa forma, o *framework* tem ponteiros para os objetos principais da aplicação – Application, Document e AppUI, que foram obtidos através de uma estrutura específica na qual a aplicação foi construída. Estes ponteiros possibilitam ao *framework* chamar funções dessas classes:

- Para descobrir a UID3 e identificar a aplicação, no caso da Application;
- Abrir ou modificar um documento, no caso da Document;
- Ou repassar um evento do sistema, no caso da AppUI.

Esses ponteiros podem ser usados pelo *framework* para controlar a aplicação, repassando eventos específicos do sistema para a aplicação.

4.4 PROJETO DA ARQUITETURA DA APLICAÇÃO

Além do núcleo básico descrito nas seções anteriores, existe o design da arquitetura do programa. Em série 60 existem três formas diferentes de implementar essa arquitetura:

- Arquitetura tradicional.
- Arquitetura baseada em caixas de diálogo.
- Arquitetura Avkon de Troca de *Views*.

A principal diferença entre eles é a forma como manipulam os eventos e como é feita a troca de *Views* dentro do programa, além de mudar a implementação dessa *View*.

Na arquitetura tradicional, o elemento de *View* herda diretamente de CCoeControl, que funciona como uma tela em branco, podendo ser desenhada conforme a sua necessidade. Essa classe é comumente chamada *Container*. Isso permite uma grande flexibilidade. Entretanto, para fazer algo mais elaborado exigem-se muitas linhas de código, o que inviabiliza certas aplicações. O AppUI é responsável por gerenciar os eventos iniciados pelo usuário, ativando cada Container de acordo com a lógica da aplicação.

Na arquitetura baseada em caixas de diálogo, o AppUI ainda é a classe que possui os controles. A diferença é que estes controles herdam de uma série de classes específicas (que por sua vez herdam de CCoeControl). Elas implementam funcionalidades como listas, formulários, entre outros elementos gráficos pré-programados pelo sistema operacional. A vantagem é obter mais funcionalidades com menos linhas de códigos,

com uma aparência padrão do sistema operacional. A desvantagem é não ter a flexibilidade da arquitetura tradicional.

A arquitetura Avkon de troca de *views* é específica do específica para a plataforma Série 60 e foi criada principalmente para ser utilizada com aplicações com múltiplas telas e múltiplas mudanças de tela. De forma similar à estrutura tradicional, ainda existe a idéia do Container, mas existe outra classe entre o Container e o AppUI, que faz o papel de AppView e herda de CAknViewAppUI. Essa classe é responsável por ativar/desativar os Containers, de acordo com comandos vindos do AppUI, de forma que apenas uma *view* esteja ativa no momento. O CAknViewAppUI é também responsável por criar e destruir as *views*, e normalmente o faz de forma que as *views* não ativadas sejam destruídas, para economizar memória.

4.5 ARQUITETURA TRADICIONAL

Para ilustrar o uso da arquitetura tradicional foi criado um exemplo simples escrevendo "*Hello*" na tela e quando o usuário escolher o comando "*next*" no menu, a mensagem é trocado por "*World*" e o item do menu é trocado para "*back*". A tela inicial do programa no emulador é mostrada na Figura 4.3. O menu da tela inicial é mostrado na Figura 4.4

Figura 4.3 Tela Inicial do Exemplo.

Figura 4.4 Menu da Tela Inicial do Exemplo.

Se for selecionado o comando "*next*" com a *softbutton* da esquerda, vai aparecer a tela mostrada é Figura 4.5.

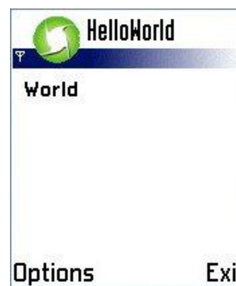

Figura 4.5 Tela Final do Exemplo.

O menu da tela final é mostrado na Figura 4.6. Se o comando "*back*" for selecionado, o programa voltará para a tela da Figura 4.3

Figura 4.6 Menu da Tela Final do Exemplo.

Além das classes mostradas anteriormente, existem ainda outros elementos que compõem a aplicação, como o arquivo de recursos, os arquivos de internacionalização, e os arquivos de definição de componentes e especificações de projeto. Esses elementos serão explicados usando o próprio código da aplicação criada nas seções seguintes

4.6 ARQUIVO DE RECURSOS

O arquivo de recursos é um arquivo texto com a extensão .rss e é usado para especificar elementos visíveis para o usuário. Outra função é especificar layout de elementos como menus, caixa de diálogos e outros. A sintaxe é similar a de um arquivo C++ , mas não é idêntico. Os comentários são no mesmo formato e as declarações de pré-processamento como #include, #define, #if,#else e #endif são aceitas. O arquivo é dividido em cabeçalho e corpo.

O cabeçalho do arquivo de recurso do exemplo *HelloWorld* é mostrado na Listagem 4.7. Primeiro é o campo NAME, um identificador de quatro letras, que não deve ser igual a outro arquivo de recurso do programa ou do sistema. A área de include funciona como em um arquivo C ou C++. O campo RSS_SIGNATURE pode ser deixado em branco, pois seu conteúdo é ignorado mas ele deve existir no arquivo para não causar um erro de compilação.

Listagem 4.7 Cabeçalho do Arquivo de Recurso (HelloWorld.rss).

```
// Identificador do recurso
NAME    HWRD

// INCLUDES

#include <avkon.loc>
#include <avkon.rsg>
#include <avkon.rh>
#include <avkon.mbg>
#include <eikon.rh>
```

```
#include "HelloWorld.hrh"
#include "HelloWorld.loc"

RESOURCE RSS_SIGNATURE { }

RESOURCE TBUF { buf=""; }
```

4.6.1 Definir um menu utilizando o Arquivo de Recurso

Para definir o menu mostrado nas Figura 4.4 e Figura 4.6 e quais itens aparecem nesse menu, é utilizado o recurso denominado MENU_BAR mostrado na Listagem 4.8.

Listagem 4.8 Declaração do Menu no Arquivo de Recurso (HelloWorld.rss).

```
RESOURCE MENU_BAR r_helloworld_menubar
   {
   titles =
      {
      MENU_TITLE
                          {
                          txt="";
                          menu_pane = r_helloworld_menu;
                          }
      };
   }
```

Normalmente um MENU_BAR contém um único MENU_TITLE. Ele especifica o título do menu e o nome do recurso que vai definir os itens desse menu. No Série 60, o item txt que define o título não é utilizado e pode ser deixando em branco. O MENU_PANE contém os itens do menu como mostra a Listagem 4.9.

Listagem 4.9 - Definição dos itens do menu (HelloWorld.rss).

```
RESOURCE MENU_PANE r_helloworld_menu
   {
   items =
      {
      MENU_ITEM
                          {
                          command = EHelloWorldProximo;
                          txt = COMMAND_NEXT;
                          },

      MENU_ITEM
                          {
                          command = EHelloWorldVoltar;
                          txt = COMMAND_BACK;
                          }
      };
   }
```

Nesse menu existem dois itens. O primeiro avança e o outro volta à tela. O MENU_ITEM tem dois campos, o nome no campo “txt” e o ID do comando. Os comandos são definidos em arquivos separados com a terminação .hrh (mostrado na Listagem 4.11). Além disso, é necessário associar esse menu à tela que será mostrada ao usuário. Isso é feito pelo recurso EIK_APP_INFO como mostra a Listagem 4.10.

Listagem 4.10 - Associação do menu a tela do usuário (HelloWorld.rss).

```
RESOURCE EIK_APP_INFO
{
```

```
    menubar = r_helloworld_menubar;
    cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT;
}
```

Para não haver conflitos com os IDs dos comandos do sistema, é recomendável que os ID definidos pelo usuário comecem em 0x6000 como mostra na Listagem 4.11.

Listagem 4.11 - Arquivo de definição dos IDs dos comandos do menu (HelloWorld.hrh).

```
#ifndef __HELLOWORLD_HRH__
#define __HELLOWORLD_HRH__

enum THelloWorldCommandIds
    {
    EHelloWorldProximo = 0x6000,
    EHelloWorldVoltar = 0x6001
    };

#endif // __HELLOWORLD_HRH__
```

4.6.2 Manuseando os Comandos

Os menus são criados nos arquivos .rss mas as ações são tomadas pelas funções:

- HandleCommandL(), nas classes CAknAppUI ou derivadas de CAknView() e;
- ProcessCommandL(), nas classes derivadas de CAknDialog().

A Listagem 4.12 apresenta um exemplo de implementação de como manipular os eventos do menu, mostrado nas Listagem 4.8 e Listagem 4.9. A implementação é basicamente apenas um *switch* que usa os ID como parâmetro e encontra-se na classe AppUI da aplicação. Além dos dois itens do menu, que são acionados pela *softbutton* da esquerda, é necessário manipular o evento de saída, gerado pelo *softbutton* da direita (EAknSoftkeyExit). Outro evento obrigatório a ser manipulado é o EEikCmdExit. Esse evento é gerado pelo sistema operacional quando se deseja fechar a aplicação.

Listagem 4.12 - Código para manipular os eventos do menu. (HelloWorldAppUi.cpp).

```
void CHelloWorldAppUi::HandleCommandL(TInt aCommand)
        {
        switch (aCommand)
                {
     case EEikCmdExit:
     case EAknSoftkeyExit:
                        {
                        Exit();
                        break;
                        }
     case EHelloWorldProximo:
        {
                iAppContainer->MudaTexto(R_LABEL_TEXT2);
                flag=1;
                        break;
        }
     case EHelloWorldVoltar:
        {
                iAppContainer->MudaTexto(R_LABEL_TEXT);
                flag=0;
                        break;
        }
        }
```

A ação EHelloWorldProximo muda o texto mostrado na tela e muda o valor da *flag* que define quais itens do menu devem aparecer. De forma similar, o EHelloWorldVoltar faz a mesma coisa, mas com diferentes valores para as duas ações.

4.6.3 Menus Dinâmicos

Outra funcionalidade dos menus é a possibilidade de mostrar ou esconder dinamicamente os itens. A Listagem 4.13 mostra como foi feito para esconder o item "*back*" da tela da Figura 4.4 e esconder o "*next*" da tela da Figura 4.6 e como foi usado a variável *flag* da Listagem 4.12 para controlar tudo isso. O parâmetro aResourceId de entrada deve ser usado para saber qual recurso (no caso o menu) deve ser alterado. A função SetItemDimmed() é responsável por definir a visibilidade dos itens e tem dois parâmetros de entrada. O primeiro é o ID do comando que vai ser selecionado e o segundo é um booleano, que deve ser ETrue para esconder o item ou EFalse para mostrar o item.

Listagem 4.13 - Código para criar um menu dinâmico (HelloWorldAppUi.cpp).

```
void CHelloWorldAppUi::DynInitMenuPaneL(TInt aResourceId, CEikMenuPane*
aMenuPane)
        {
          if (aResourceId == R_HELLOWORLD_MENU)
                {
                  if (flag == 0 )
                        {
                          aMenuPane->SetItemDimmed(EHelloWorldProximo, EFalse);
                          aMenuPane->SetItemDimmed(EHelloWorldVoltar, ETrue);
```

```
                         }
        else
        {
           aMenuPane->SetItemDimmed(EHelloWorldProximo, ETrue);
                        aMenuPane->SetItemDimmed(EHelloWorldVoltar, EFalse);
        }
                  }

        }
```

4.6.4 String no Arquivo de Recurso e Internacionalização

A forma mais recomendada de inserir textos na sua aplicação é utilizar a conhecida técnica de internacionalização. As línguas disponíveis devem ser agrupadas em um arquivo .loc e deve ser incluído no cabeçalho do arquivo de recursos como se mostra na Listagem 4.7 (#include "HelloWorld.loc"). Dentro do arquivo .loc coloca-se um #include para um arquivo tipo .lxx onde xx é o código de dois dígitos da linguagem, definido no arquivo e32std.h. Um exemplo de arquivo .loc é mostrado na Listagem 4.14. Os textos em inglês do arquivo HelloWorld.l01 são mostrados na Listagem 4.15.

Listagem 4.14 - Arquivo com as *strings* para cada língua (HelloWorld.loc).

```
/ arquivo de linguagem do inglês britânico
#ifdef LANGUAGE_01
#include "HelloWorld.l01"
#endif
// arquivo de linguagem do português do Brasil
#ifdef LANGUAGE_76
#include "HelloWorld.l76"
#endif
```

Listagem 4.15 - Texto da língua inglesa para a aplicação (HelloWorld.l01).

```
/ texto da Aplicação
#define LABEL_TEXT "Hello"
#define LABEL_TEXT2 "World"

// texto do Menu.
#define COMMAND_NEXT "Next"
#define COMMAND_BACK "Back"
```

É necessário criar um recurso denominado TBUF para utilizar *strings* definidas no arquivo HelloWorld.l01 dentro do aplicativo. A sintaxe no arquivo de recurso é mostrada o trecho da Listagem 4.16.

Listagem 4.16 - Inserindo uma string como recurso (HelloWorld.rss).

```
RESOURCE TBUF r_label_text
        {
        buf = LABEL_TEXT;
        }
```

Dentro do aplicativo, r_label_text deve ser usado com caixa alta em todo o nome. Por exemplo, o do parâmetro de entrada da função MudaTexto() na Listagem 4.12.

O texto do menu pode ser usado diretamente como se mostra na Listagem 4.9 e não é necessário criar um recurso TBUF .

4.6.5 Função para Mudar o Texto

A função para mudar o texto na tela da Figura 4.3 para a Figura 4.5 é mostrado na Listagem 4.17.

Listagem 4.17 - Função para mudar o texto na tela (HelloWorldContainer.cpp).

```
void  CHelloWorldContainer::MudaTexto(TInt buffer)
      {
       HBufC* labelText;
          labelText = StringLoader::LoadLC(buffer);
          iLabel->SetTextL(*labelText);
          CleanupStack::PopAndDestroy(labelText);
       SizeChanged();
      }
```

Basicamente usa-se o ID de um recurso de texto fornecido como parâmetro de entrada (mostrado com mais detalhes no item de internacionalização). Cria-se um objeto para converter o recurso em uma string que pode ser usado pelo objeto iLabel. Após a conversão, mostra o texto na tela. Depois deve usar a função SizeChanged() para mudar o tamanho que deve ter esse iLabel na tela.

4.7 GERANDO O PROJETO

Para gerar o projeto, inicialmente, é necessário criar o arquivo de definição de componentes, bld.inf. Este arquivo não tem nenhuma mudança com relação ao descrito para a aplicação em texto. O código pode ser visto na Listagem 4.18. Este arquivo apenas aponta para o arquivo de especificações de projeto (HelloWorld.mmp). O código do arquivo mmp pode ser visto na Listagem 4.19.

Listagem 4.18 - Código do arquivo bld.inf.

```
PRJ_MMPFILES
HelloWorld.mmp
```

Listagem 4.19 - Código do arquivo HelloWorld.mmp.

```
TARGET  HelloWorld.app
TARGETTYPE  app
UID  0x100039CE 0x101FDA4D
TARGETPATH \system\apps\HelloWorld

SOURCEPATH ..\src
SOURCE  HelloWorldApplication.cpp
SOURCE  HelloWorldAppUi.cpp
SOURCE  HelloWorldDocument.cpp
SOURCE  HelloWorldContainer.cpp

LANG 01
```

```
RESOURCE ..\data\HelloWorld.rss
RESOURCE ..\data\HelloWorld_caption.rss

USERINCLUDE .
USERINCLUDE ..\src

SYSTEMINCLUDE \epoc32\include

LIBRARY euser.lib apparc.lib cone.lib eikcore.lib
LIBRARY eikcoctl.lib avkon.lib commonengine.lib

AIF HelloWorld.aif ..\aif HelloWorldAif.rss c12 context_pane_icon.bmp
context_pane_icon_mask.bmp list_icon.bmp list_icon_mask.bmp
```

Este arquivo foi explicado no Capítulo 3, para uma aplicação baseada em console. É fácil ver que esta aplicação inclui mais arquivos de código-fonte e bibliotecas, além de arquivos de recursos.

Com relação ao alvo (*target*) e ao tipo de alvo (*targetype*), ambos denotam uma *application (app).* Isso define que esta aplicação tem uma interface gráfica, definindo o número do UID1. O campo UID tem dois números: o primeiro, 0x100039CE indica que esta é uma aplicação *app* com interface gráfica, e o segundo é o UID3 da aplicação, que define unicamente esta aplicação Symbian. O valor usado para o UID3, 0x012233ED foi escolhido dentro da gama de números de teste – este é o valor a ser requisitado para a Symbian caso esta fosse uma aplicação comercial.

O outro campo diferente é o LANG – já explicado em internacionalização. O último campo diferente é o AIF – de *Application Information* (informações da aplicação). Este campo define onde estarão os arquivos usados em tempo de execução, como ícones e imagens.

4.8 CONCLUSÃO DO CAPITULO

Com o conhecimento como funciona a interface gráfica do Symbian C++, vários tipos de aplicações já pode ser feitas. Infelizmente, algumas bibliotecas exigem que sejam chamadas de forma assíncrona. A implementação dessa funcionalidade é mostrada no próximo capítulo.

5 MULTI TAREFA

O Symbian OS é um sistema com suporte a operações assíncronas. Geralmente, utilizam-se multiprocessos ou em sistema moderno *multithread*. Mas, no Symbian, existe um sistema diferenciado chamado de *multitasking* cooperativo. Esse sistema exige que a tarefa em execução ceda o processador a outras tarefas a serem executadas.

5.1 MULTITASKING COOPERATIVO

A implementação envolve dois elementos, um temporizador e os objetos ativos. O temporizador é um *loop* de espera que detecta os objetos completos e retorna às suas respostas. O objeto ativo encapsula uma requisição do usuário em um formato que o temporizador possa manipular.

5.2 ACTIVE SCHEDULER

O temporizador em Symbian OS é implementado por CActiveScheduler ou por uma classe derivada. Ela implementa o *loop* de espera em uma única *thread*, que detecta o término de eventos assíncronos e localiza o objeto ativo correspondente. Esse objeto mantém uma lista de todos os objetos ativos do sistema. Os objetos completados são detectados pela chamada da função Use::WaitForAnyRequest(). Esse controle é feito usando-se duas *flags*. A *flag* iActive determina se um pedido foi feito, verificando se essa *flag* está ativada. A segunda iStatus determina a condição do objeto. Qualquer valor diferente de KRequestPending é considerado uma requisição completada.

5.3 ACTIVE OBJECTS

Esse objeto representa a requisição assíncrona de um serviço. Normalmente ele encapsula

- Estado da requisição (iStatus)
- Manipular a requisição após estar completada (RunL())
- Função para cancelar uma requisição (Cancel())
- Função para emitir ou re-emitir uma requisição assíncrona (nome da função definida pelo usuário e normalmente é denominada Start())

A definição do objeto ativo é mostrada Listagem 5.1. Para entender como fazer um objeto ativo, cada método do objeto Cative vai ser explicado abaixo.

Listagem 5.1 – Funções do CAtive

```
class CActive : public CBase
{
public:
   virtual ~CActive();
   void Cancel();
   inline TBool IsActive() const;

protected:
   CActive(TInt aPriority);
   void SetActive();
   virtual void DoCancel() =0;
  virtual void RunL() =0;
   virtual TInt RunError(TInt aError);


public:

    TRequestStatus iStatus;

private:
    TBool iActive;
    friend class CActiveScheduler;
};
```

Os métodos públicos do objeto CActive são:

~CAtive() – O destrutor público sempre deve chamar o método Cancel() para garantir que todas as requisições feitas pelo objeto serão destruídas.

Cancel() – Método público para cancelar as requisições, chama obrigatoriamente a função protegia Docancel(). Além disso, é recomendado sempre chamar Cancel() e não o DoCancel().

IsActive() – Função que determina se o objeto está ativo.

Outro método público sem nome definido é para emitir ou re-emitir uma requisição. Normalmente o nome é start(), mas este depende da implementação do objeto ativo.

Os Métodos Protegidos do objeto CActive são:

SetActive() – método que deve ser chamado quando uma requisição assíncrona for emitida e para que o temporizador (CActiveScheduler) localize esse objeto quando estiver terminado.

DoCancel() – esse método é puramente virtual, ou seja, a implementação dele é obrigatória. O único método que deve chamá-lo é o Cancel(). Existem controles feitos somente pelo método Cancel(). Portanto, utiliza-se somente o método Cancel().

RunL() – outro método que deve ser implementado obrigatoriamente. Quando uma requisição estiver completa, o CActiveScheduler chama esse método. O comum é encontrar uma máquina de estados. Exemplos disso serão mostrados no próximo Capítulo. É muito importante alertar que, dentro de uma mesma *thread*, o ideal é manter o processamento do Runl() no menor tempo possível. Se esse método for muito demorado, pode parecer para o usuário, que o programa travou-se.

O variável membro iStatus é o controle do sistema operacional de como deve proceder com o objeto ativo. O estado ativado é KResquestPending. Quando uma requisição for completada, o valor muda para KErrNone ou um valor de erro.

5.3.1 Implementando um objeto Ativo

Existe uma série de passos para criar um objeto ativo:

- Criar uma classe que herda de CActive.
- Invocar o construtor de CActive com a prioridade desejada. O normal é EPriorityStandard.
- Invocar o método CActiveScheduler::Add() na construção do objeto.
- Implementar uma função para iniciar o objeto, que normalmente é chamado de Start(). É importante lembrar, essa função deve chamar SetActive() na sua implementação para ativar o objeto.
- Implementar a função RunL() para manipular o objeto após a requisição feita no item anterior estiver completa.
- Implementar as funções DoCancel() e Cancel() para cancelar as requisições desse objeto e destruí-lo.

5.4 CONCLUSÃO DO CAPITULO

O entendimento dos objetos ativos é importante, pois várias funções do sistema devem ser feitas dentro de forma assíncrona. O próximo capítulo mostrará a comunicação bluetooth que faz uso desses objetos ativos para criar um servidor que anuncia os serviços e cria a conexão com os clientes.

6 COMUNICAÇÃO EM SYMBIAN C++

Este Capítulo vai mostrar alguns recursos que o Symbian OS oferece ao programador para essa interação com outros dispositivos. Uma ênfase no protocolo bluetooth é dada, pois é a forma de comunicação entre o Nokia 6600 e o receptor GPS.

6.1 TIPOS DE COMUNICAÇÃO

Os protocolos de comunicação disponíveis em Symbian OS são:

- Serial – essa comunicação é simples e feita para conectar dois dispositivos próximos. Isso pode ser feito por infravermelho ou Bluetooth
- *Socket* – comunicação ponto a ponto. Isso pode ser feito entre uma rede física ou lógica. Os meios possíveis são infravermelho, Bluetooth ou *TCP/IP*

Existem três meios disponíveis no Symbian OS:

- TCP/IP – protocolo de comunicação usado na Internet, utilizando o *socket* para acessar os serviços disponíveis por meio da rede de pacotes que o sistema celular fornece. Um exemplo é o GPRS na rede GSM.
- Infravermelho – sistema de comunicação ponto-a-ponto de pequeno alcance, que precisa ter linha de visão entre os dispositivos.
- Bluetooth – Uso parecido com o infravermelho. Utilizando a tecnologia de rádio-freqüência para essa comunicação.

6.2 BLUETOOTH

Bluetooth é uma comunicação sem fio de pequeno alcance que opera em torno de 2,45 GHz. A sua conexão pode ser feita ponto-a-ponto ou multiponto. A pilha de protocolos do Bluetooth é constituída de:

- Camada física
- Protocolo Bluetooth – camada de controle da rede com uma função similar ao TCP/IP.
- *Profile* Bluetooth – camada dos serviços disponíveis

Os protocolos Bluetooth aceitos em Symbian OS são:

- *Link Manager Protocol* (LMP) – controla o comportamento da camada física dos *links* sem fio e permite o equipamento ser descoberto.
- L2CAP - controla a transmissão de dados com conexão e sem conexão.

- RFCOMM – Protocolo de transporte que emula o comportamento da RS-232.
- SDP – Protocolo de localização para que o equipamento possa encontrar os outros dispositivos próximos.

Os perfis são como os serviços disponíveis para o usuário utilizar no dispositivo. O conjunto de perfis é definido pelo fabricante, não existindo um conjunto obrigatório para um celular da serie 60.

- *Object Push* – permite o bluetooth enviar objetos definidos pelo protocolo OBEX.
- Transferência de arquivo – permite navegar, deletar ou enviar arquivos pelo bluetooth.
- DUN (*dial-up network*) – utilizar o telefone como modem
- Porta serial – Isso dá suporte a simular uma conexão RS-232

6.2.1 Anúncio e descoberta de serviços em Bluetooth

Um dispositivo, para ser encontrado por outros, deve anunciar seus serviços disponíveis. Isso é feito pelo SDD (*Service Discovery DataBase*) que é uma base de dados dos serviços que onde cada entrada é composta por quatro elementos:

- UUID – Número único que identifica o serviço a ser usado
- Porta – Porta onde o serviço registrado vai aceitar novas conexões
- Classe ID – número para identificar o tipo de serviço oferecido
- Disponibilidade – Controle para definir a disponibilidade de um serviço na base de dados.

Para acessar a SDD em Symbian deve-se usar os objetos RSdp e RSdpDataBase

A Listagem 6.1 mostra um exemplo de código para fazer essa conexão:

Listagem 6.1 Conectando na base de dados (SDD).

```
TBool iIsConnected;

RSdp iSdpSession;

RSdpDatabase iSdpDatabase;

User::LeaveIfError(iSdpSession.Connect());
User::LeaveIfError(iSdpDatabase.Open(iSdpSession));
iIsConnected = ETrue;
```

Se a conexão for criada com sucesso, deve-se criar um novo registro na base de dados. O primeiro passo é criar um UUID que identifica seu serviço. Pode-se usar o UID3

(visto em 3.5) da aplicação para isso. Além disso, existem algumas normas de UUID[12] que fogem do escopo desta documentação.

A Listagem 6.2 mostra como inserir uma UUID nova usando a conexão com o SDD, mostrada na Listagem 6.1. O retorno da função CreateServiceRecordL é armazenado em variável do tipo TSdpServRecordHandle.

Listagem 6.2 criando uma nova entrada no SDD.

```
const TUint KUUID = 0x101F6148
TSdpServRecordHandle iRecord;

iSdpDatabase.CreateServiceRecordL(KUUID, iRecord);
```

Em seguida, o objeto iRecord representa o serviço que vai ser anunciado. Em Symbian, o serviço criado pode ser visto com um servidor TCP/IP. A Listagem 6.3 mostra como criar esse servidor para as conexões via bluetooth. [14]

Listagem 6.3 Criando o servidor *socket*

```
RSocketServ iSocketServer;
RSocket iListeningSocket;
User::LeaveIfError(iSocketServer.Connect());
User::LeaveIfError(iListeningSocket.Open(iSocketServer, "RFCOMM"));
```

Com o *socket* criado com sucesso, torna-se necessário obter uma porta para publicar na SDD. A Listagem 6.4 mostra como é gerada a variável iChannel que armazena esse dado. [14]

Listagem 6.4 Obtendo a porta disponível.

```
TInt iChannel;
User::LeaveIfError(iListeningSocket.GetOpt(KRFCOMMGetAvailableServerChannel,
KSolBtRFCOMM, iChannel));
```

Com o *socket* criado e obtido a porta, é necessário atualizar o registro criado em Listagem 6.2 com esses novas informações. Isso é obtido construindo-se um *data Element Sequence* (DES) usando o objeto CsdpAttrValueDES.[15]. A Listagem 6.5 mostra como popular o objeto aProtocolDescriptor e como associar ao registro na SDD (iRecord). Além disso, avisa que essa entrada está pronta.

Listagem 6.5 Configurando a entrada do SDD.

```
CSdpAttrValueDES* aProtocolDescriptor
TBuf8<1> channel;
        channel.Append((TChar)aPort);

        aProtocolDescriptor
        ->StartListL()     // List of protocols required for this method
                ->BuildDESL()
                ->StartListL()     // Details of lowest level protocol
                        ->BuildUUIDL(KL2CAP)
                ->EndListL()

                ->BuildDESL()
                ->StartListL()
                        ->BuildUUIDL(KRFCOMM)
                        ->BuildUintL(channel)
```

```
                ->EndListL()
        ->EndListL();
iSdpDatabase.UpdateAttributeL(iRecord,KSdpAttrIdProtocolDescriptorList,
*aProtocolDescriptor);
TBool aIsAvailable = ETrue;
if (aIsAvailable)
{
state = KStateFullyUnused; // disponivel para conexão
}
else
{
state = KStateFullyUsed;        // indisponivel
}

iSdpDatabase.UpdateAttributeL(iRecord, KSdpAttrIdServiceAvailability, state);
```

O último passo é necessário conhecer os Objetos Ativos (veja no capítulo 5). Dentro de um objeto do tipo Cative, o servidor criado em Listagem 6.3 deve estar preparado para uma nova conexão e isso deve ser feito usando-se chamada assíncrona. Esse processo é mostrado em Listagem 6.6.

Listagem 6.6 Configurando o servidor *socket* para novas conexões.

```
TBTSockAddr listeningAddress;
RSocket iAcceptedSocket;
listeningAddress.SetPort(channel);

User::LeaveIfError(iListeningSocket.Bind(listeningAddress));
User::LeaveIfError(iListeningSocket.Listen(KListeningQueSize));

iAcceptedSocket.Close();
User::LeaveIfError(iAcceptedSocket.Open(iSocketServer));

iListeningSocket.Accept(iAcceptedSocket, iStatus);
SetActive();
```

6.2.2 Procurando e conectando a um dispositivo bluetooth

A seção 6.2.1 mostrou como criar um serviço bluetooth, Esta seção vai mostrar como encontrar o dispositivo e quais serviços estão disponíveis. A classe utilizada para encontrar esses dispositivos é RNotifier. Esse objeto deve ser inicializado com o comando Connect(). Essa classe tem muitas funções, que possibilitam mostrar caixas de diálogo para o usuário. O comportamento é definido por um ID único (UID) na entrada da função. Para a seleção de dispositivos bluetooth é usado o valor KDeviceSelectionNotifierUid. A função usada é StartNotifierAndGetResponse(). Um exemplo para configurar uma busca filtrada a fim de encontrar somente os dispositivos desejados é mostrado em Listagem 6.7 . Infelizmente, o filtro não funciona na prática, todos os dispositivos são mostrados na busca. Os parâmetros de entrada da função StartNotifierAndGetResponse() são:

- iStatus é variável de controle do objeto ativo do tipo TRequestStatus&
- aResponse é variável do tipo TBTDeviceResponseParamsPckg que armazena a resposta da função

Listagem 6.7 Busca de dispositivos bluetooth.

```
{
TBTDeviceResponseParamsPckg      aResponse;
RNotifier                        iNotifier;
TUUID targetServiceClass(KServiceClass);
TBTDeviceClass deviceClass(KServiceClass);
TBTDeviceSelectionParams selectionFilter;
selectionFilter.SetUUID(targetServiceClass);
selectionFilter.SetDeviceClass(deviceClass);

TBTDeviceSelectionParamsPckg selectionParams(selectionFilter);

User::LeaveIfError(iNotifier.Connect());

iNotifier.StartNotifierAndGetResponse(iStatus, KDeviceSelectionNotifierUid,
selectionParams, aResponse);
SetActive();
}
```

Se a buscar ocorrer com sucesso e for um dispositivo válido, torna-se necessário verificar os serviços disponíveis. Esse processo vai verificar se existe o serviço desejado na SDD e qual porta está associada ao serviço. O primeiro passo é obter o endereço representado pelo objeto TBTDevAddr que é obtido do retorno de StartNotifierAndGetResponse()

Três classes são usadas para a procura do serviço

- A classe MSdpAgentNotifier com três métodos puramente virtuais:
  - NextRecordRequestCompleteL(TInt aError, TSdpServRecordHandle aHandle, TInt aTotalRecordsCount) para
  - AttributeRequestResultL(TSdpServRecordHandle aHandle, TSdpAttributeID aAttrID, CSdpAttrValue* aAttrValue);
  - AttributeRequestCompleteL(TSdpServRecordHandle aHandle, TInt aError);
- A classe CSdpAgent para fazer a procura que na sua criação precisa de um objeto MSdpAgentNotifier para comunicar os resultados
- A classe CSdpSearchPattern para definir os parâmetros da procura do objeto CSdpAgent, que é associado pela função CSdpAgent::SetrecordFilterL()

A Listagem 6.8 mostra o inicio da pesquisa feita pelo UUID que foi definido na Listagem 3.1 com identificador do serviço anunciado. Além disso, usa-se o ponteiro

*this, pois esse código é chamado dentro do objeto que implementa MSdpAgentNotifier.

Listagem 6.8 Iniciando uma procura pelo serviço dentro de MSdpAgentNotifier

```
const TUint KUUID = 0x101F6148;
TBTDevAddr aDeviceAddress = aResponse.BDAddr();
CSdpSearchPattern * iSdpSearchPattern = CSdpSearchPattern::NewL();
iSdpSearchPattern->AddL(KUUID);
CSdpAgent  * iAgent = CSdpAgent::NewL(*this, aDeviceAddress);
iAgent->SetRecordFilterL(*iSdpSearchPattern);
iAgent->NextRecordRequestL();
```

Quando a função NextRecordRequestL() for completada, a função NextRecordRequestCompleteL do MSdpAgentNotifier é chamada. As entradas dessa função são: o código de erro, um manipulador de evento e o numero de registros encontrados pelos parâmetros do filtro.

Um exemplo de implementação de NextRecordRequestCompleteL é mostrado em Listagem 6.9

Basicamente, dentro da função, cria-se uma lista de informações que deve ser preenchida pelo registro encontrado na Listagem 6.8.

Listagem 6.9 Definição do método NextRecordRequestCompletL

```
// variáveis criadas no .h
CSdpAttrIdMatchList*    iMatchList;

void CBluetoothServiceSearcher::NextRecordRequestCompleteL(TInt aError,
TSdpServRecordHandle aHandle, TInt aTotalRecordsCount)
        {
                if (aError == KErrNone && aTotalRecordsCount > 0)
                {
                iContinueSearching = ETrue; // Reset for this record
                iMatchList = CSdpAttrIdMatchList::NewL();
                iMatchList->AddL( KSdpAttrIdServiceAvailability );
                iMatchList->AddL( KSdpAttrIdProtocolDescriptorList );


                iAgent->AttributeRequestL(aHandle, *iMatchList);
                }

        }
```

Seguindo uma lógica similar quando a função iAgent->AttributeRequestL(aHandle, *iMatchList) for completada, a função AttributeRequestResultL é chamada. Um exemplo de implementação é mostrada em Listagem 6.10. Os dados requisitados são identificados por um ID (TSdpAttributeID aAttrID) e o seu valor é armazenado em um ponteiro CSdpAttrValue* aAttrValue. Pelo exemplo da Listagem 6.9, foi pedido dois dados:

um é a lista de protocolos definidos com o ID de KSdpAttrIdProtocolDescriptorList e o outro é a disponibilidade do serviço com o ID de KSdpAttrIdServiceAvailability.

Listagem 6.10 Definição do método AttributeRequestResultL

```
void CBluetoothServiceSearcher::AttributeRequestResultL(TSdpServRecordHandle ,
TSdpAttributeID aAttrID, CSdpAttrValue* aAttrValue)
	{
	if (aAttrID == KSdpAttrIdProtocolDescriptorList)
		{

		TBluetoothAttributeParser parser(*this, iContinueSearching);
		aAttrValue->AcceptVisitorL(parser);
		}
	else if (aAttrID == KSdpAttrIdServiceAvailability)
		{
		if (aAttrValue->Type() == ETypeUint)
			{
			iAvailable = static_cast<TBool>(aAttrValue->Uint());
				if (iAvailable)
				{
					iSearcherState = KErrNone;

				}
			}
		}
	delete aAttrValue;
	}
```

Para obter os dados individuais da estrutura KSdpAttrIdProtocolDescriptorList deve-se estender o objeto MSdpAttributeValueVisitor onde o principal método é VisitAttributeValueL. Um exemplo é mostrado na Listagem 6.11. O resultado disso é a porta do serviço. Agora com essa informação, será criado o *socket* para comunicação entre os dispositivos.

Listagem 6.11 Definição do método VisitAttributeValueL

```
void TBluetoothAttributeParser::VisitAttributeValueL(CSdpAttrValue& aValue,
TSdpElementType /*aType*/)
	{
	switch ( iProcessingState )
		{
		case EStartOfDesOne:
		case EStartOfDesTwo:
		case EL2Cap:
		case EEndOfDesTwo:
		case EStartDesThree:
			break; // check nothing

		case ERFComm:
			CompareRFCOMM( aValue );
			break;

		case ERFCommPort:
			GetPort( aValue );
			break;

		case EEndOfDesThree:
		case EEndOfDesOne:
			break; // check nothing

		default: // unexpected output -- panic
			Panic(EBadAttributeValue);
			break;
		}

	Next();
	}
```

6.2.3 Comunicação *socket* entre os dispositivos

O servidor é o dispositivo que anunciou o serviço. O processo de anunciar foi mostrado no item 6.2.1. O cliente é o dispositivo que procurou o serviço. Essa seção vai mostrar o que o servidor deve fazer quando o cliente conectar com sucesso. Além de, mostrar os procedimentos relativos ao cliente.

6.2.3.1 Servidor

Na Listagem 6.6, o servidor foi configurado para esperar uma conexão dentro de um objeto ativo. O método RunL() é mostrado na Listagem 6.12 e tem dois estados possíveis. O primeiro ocorre quando a conexão é criada com o cliente e serviço deixa de ser anunciado na SDD, pois somente um cliente pode ser atendido. O Symbian é limitado a somente um cliente. O objeto para a comunicação é o RSocket. Para maiores informações sobre o objeto RSocket veja a documentação [23] . O segundo passo consiste em manipular o resultado do estado anterior, que novamente é dependente da função do servidor.

Listagem 6.12 Manipulação do *socket* no servidor.

```
void RunL(){
switch (iEstado)
{
	case EConectado:
		iAdvertiser->StopAdvertisingL();
		// Manipulação do socket representado pelo socket
		//Ler os dados enviados pelo cliente
		iAcceptedSocket.RecvOneOrMore(iBuffer, 0, iStatus, iLen);
		//Mudar o estado para ler o buffer quando o RunL for chamdo
novamente
		iEstado = ERecebido;
		// Ativa novamente o objeto ativo
		setActive();
	case ERecebido:
		// o buffer foi prenchido e pode ser manipulado
		// ...

}
```

6.2.3.2 Cliente

O cliente tem, após a procura feita na Listagem 6.11, a porta do serviço anunciado no servidor e o endereço do servidor obtido na Listagem 6.7. A criação do *socket* de comunicação é feita no exemplo da Listagem 6.13. Isso deve ser feito em um objeto ativo. O método `RunL()` pode ser feito de forma similar ao da Listagem 6.12.

Listagem 6.13 Criando o *socket* no cliente com o servidor.

```
iEstado  = EConectado;

// os dados do endereço e da porta vindo da Listagem 6.7 e Listagem 6.11
repectivamente
	iSocketAddress.SetBTAddr(aBTDevAddr);
	iSocketAddress.SetPort(aPort);

	User::LeaveIfError(iSocketServer.Connect());
	User::LeaveIfError(iSendingSocket.Open(iSocketServer, KServerTransportName));

	iSendingSocket.Connect(iSocketAddress, iStatus);

	SetActive();
```

6.3 CONCLUSÃO DO CAPITULO

O entendimento da implementação da comunicação Bluetooth desse capítulo é a base para obter os dados do hardware externo GPSSlim236. O próximo capítulo vai mostrar como obter os dados do telefone de várias formas diferentes dependente da versão do sistema operacional. Além de, fazer uma análise dos pontos fortes e fracos desses métodos para obter os dados do telefone.

7 INFORMAÇÕES SOBRE O DISPOSITIVO

Algumas informações sobre a rede GSM e o celular são possível de serem obtidas dentro do Symbian C++. Nesse capítulo, quatro informações serão usadas como exemplo para demonstrar os métodos de obtenção. Uma breve explicação dessas informações é mostrada abaixo:

- Potência do sinal recebido.
- IMSI (International Mobile subscriber Identity). é o numero indentificador do usuário dentro da rede GSM que é formado por:
    - MCC = Mobile Country Code
    - MNC= Mobile Network Code
    - MSIN= Mobile Subscriber ID. Number
- IMEI (International Mobile Equipment Identity), é o numero de serie do aparelho, para identificar a validade desse aparelho dentro do *Equipment Identity Register* (EIR) por meio de tres listas:
    - : *White List* - operação normal
    - *Grey List* – operação monitorada
    - *Black List* – operação proibida
- *Cell Id é o numero indentificador da celula que o usuario está registrado no momento.*

O método mais antigo, que funciona desde a versão 6.1 do sistema operacional, usa a estrutura RBasicGsmPhone [5]. O segundo método usa a biblioteca mobinfo [4] compatível com as versões v7.0, v7.0s, 8.0a e v8.1a. Para as versões v9 em diante usa-se a solução padrão ETel3rdParty (CTelephony) API. Nas seções seguintes será mostrado o uso e vantagens e desvantagens dos dois primeiros métodos. Infelizmente, somente os dois primeiros métodos foram testados, porque não havia nenhum dispositivo v9 disponível para os testes.

7.1 INFORMAÇÕES DO TELEFONE USANDO RBASICGSMPHONE

O SDK da Série 60 oficialmente não suporta utilizar essa estrutura para obter esses dados. Para isso, deve-se pegar o arquivo ETELBGSM.H do Nokia 9210 *Communicator* beta SDK e colocar em um diretório de *include* do projeto. As duas bibliotecas a serem

incluídas são gsmbas.lib e etel.lib. Os códigos dessa seção vão gerar números coerentes somente em um celular real. A inicialização do objeto RBasicGsmPhone é mostrada em Listagem 7.1

Listagem 7.1 Inicialização da Estrutura RBasicGsmPhone.

```
RBasicGsmPhone phone;
RTelServer server;

        User::LeaveIfError( server.Connect() );
        // load a phone profile
        _LIT(KGsmModuleName, "phonetsy.tsy");
        User::LeaveIfError( server.LoadPhoneModule( KGsmModuleName ) );

        // initialize the phone object
        RTelServer::TPhoneInfo info;
        User::LeaveIfError( server.GetPhoneInfo( 0, info ) );
        User::LeaveIfError( phone.Open( server, info.iName ) );
```

Após essa iniciação é fácil obter as informações sobre o telefone. O arquivo ETELBGSM.H mostra, em detalhes, as estruturas internas e os métodos do objeto RBasicGsmPhone. Para obter os dados de *cell id* é necessário utilizar uma outra estrutura de dados denominada TCurrentNetworkInfo. A Listagem 7.2 mostra detalhes de como completar essa estrutura para obter o *cell id.* Para obter o sinal recebido em porcentagem é usado a função GetSignalStrength() e o IMEI é usado a estrutura Tid. Um exemplo é mostrado na Listagem 7.3.

Listagem 7.2 Obtenção do cell id

```
struct TCurrentNetworkInfo
                {
                TNetworkInfo iNetworkInfo;
                TUint iLocationAreaCode;
                TUint iCellId;
                };
        TUint cellid;
        MBasicGsmPhoneNetwork::TCurrentNetworkInfo ni;
        User::LeaveIfError( phone.GetCurrentNetworkInfo( ni ) );
        TUint cellid;
        cellid = ni.iCellId;
```

Listagem 7.3 Obtendo o IMEI.

```
struct TId
                {
                TBuf<KPhoneManufacturerIdSize> iManufacturerId;
                TBuf<KPhoneModelIdSize> iModelId;
                TBuf<KPhoneRevisionIdSize> iRevisionId;
                TBuf<KPhoneSerialNumberSize> iSerialNumber;
                };
MBasicGsmPhoneId::TId m_id;
User::LeaveIfError(phone.GetGsmPhoneId(m_id));
```

7.1.1 Vantagens do método RBasicGsmPhone

O método é o possível na versão 6.1 do Symbian OS. Não é necessário instalar nenhuma biblioteca externa ao celular. Nem é necessário utilizar objetos ativos para obter a informação, tornando o código mais simples.

7.1.2 Desvantagens do método RBasicGsmPhone

O método não tem suporte oficial em Serie 60. Assim sendo, não existe uma documentação oficial do uso dessa estrutura na plataforma.

7.2 INFORMAÇÕES DO TELEFONE USANDO MOBINFO

Para utilizar a biblioteca MOBINFO, algumas configurações devem ser feitas tanto no ambiente emulado e como no celular. Os arquivos que serão usados são o mobileinfo.h, mobinfotypes.h e mobinfo.dll. Os passos envolvidos são:

- Baixar os arquivos de [4]
- Copiar os arquivos mobinfo.lib/dll para os arquivos do SDK:
    - \epoc32\release\wins\udeb\
    - \epoc32\release\wins\urel\
    - \epoc32\release\armi\urel\
- Incluir os arquivos "mobileinfo.h", "mobinfotypes.h" nos diretórios de *include* do projeto.
- Instalar a biblioteca no celular. Incluir a linha

@"<path >\mobinfo.sis",(0x1020483d)

no arquivo .PKG do projeto.

Juntamente com os arquivos, há um arquivo de documentação [4] no formato pdf. Ele mostra como funciona a chamada de função dessa biblioteca. A Listagem 7.4 mostra a estrutura de dados e os tipos que recebem os dados de *cell id*. Mostra também a potência do sinal que é dado em forma de cinco níveis e IMEI. Esses dados foram retirados do arquivo *header* "mobinfotypes.h".

Listagem 7.4 Trecho de "mobinfotypes.h".

```
struct TMobileCellId                        //CGI
   {
   TBuf<KMobileSizeOfMCCText>     iCountryCode;      //MCC
   TBuf<KMobileSizeOfMNCText>     iNetworkIdentity;  //MNC
   TUint                    iLocationAreaCode;  //LAC
   TUint                    iCellId;          //CI
```

```
    };

const TInt         KMobileSignalStrengthMax=5;
const TInt         KMobileSignalStrengthMin=0;
typedef TInt         TMobileSignalStrength;

const TInt KMobileSizeOfIMEI = 50;
typedef TBuf<KMobileSizeOfIMEI>        TMobileIMEI;
```

As funções para obter as informações dessa biblioteca devem ser chamadas dentro de objeto ativo (veja o capitulo 5). Um exemplo de implementação de uma função RunL() que, a cada estado, completa dados do celular é mostrado em Listagem 7.5.

Listagem 7.5 Obtendo IMEI e *Cell Id* via mobinfo API.

```
// objetos definidos na biblioteca mobinfo
        CMobileInfo* mi;
        CMobileNetworkInfo *ni;
//

void CMobInfoActive::RunL()
        {
        if(state == 1)
        {

                mi->GetIMEI(imei, iStatus);
                SetActive();
        }
        else if(state == 2)
        {
                state = 3;
                iAppView->imei = imei;
                ShowText();

                ni->GetCellId(cellid, iStatus);
                SetActive();

        }
        else if(state == 3)
        {
                state = 4;
                iAppView->cellid = cellid().iCellId;
                ShowText();
        }


        }
```

7.2.1 Vantagens da MobInfo

A biblioteca MOBINFO apresenta uma solução com uma documentação bem definida, com comportamento garantido para várias versões de Symbian OS.

7.2.2 Desvantagens da MobInfo

A biblioteca MOBINFO é uma solução externa ao sistema. Ou seja, é necessária a instalação da biblioteca e necessita-se usar objeto ativo que complica um pouco o código.

7.3 CONCLUSÃO DO CAPITULO

Foi apresentado os conceitos de alguns parâmetros da rede celular e três métodos para obté-los. O próximo capítulo mostra a implementação de dois desses métodos e outros conceitos de capítulos anteriores.

8 APLICAÇÃO FINAL

A Aplicação final reúne todos os conceitos de programação mostrados nos capítulos anteriores. As funcionalidades são:

- Obter os dados GPS via bluetooth,
- Obter dados do telefone via MobInfo
- Obter dados do telefone via RBasePhone

O telefone usado foi o Nokia 6600[27] e o GPS foi o Holux GPSSlim236[26]. Um diagrama dos elementos envolvidos na aplicação é mostrado na Figura 8.1

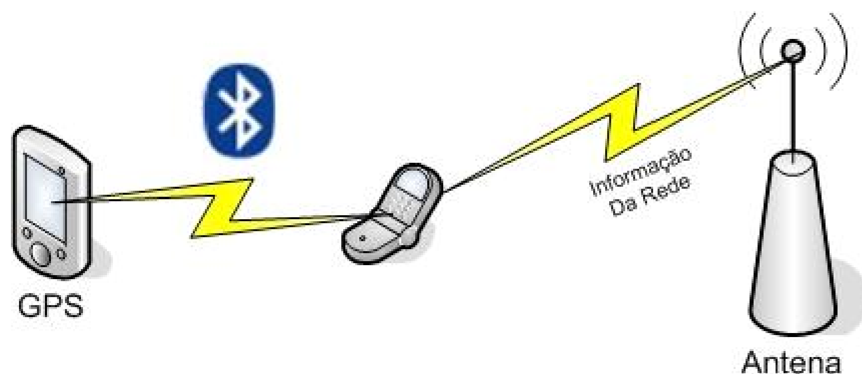

Figura 8.1 Diagrama dos elementos da aplicação final.

8.1 INTERFACE DO PROGRAMA

A interface é muito simples, existe uma tela que escreve o status e os resultados das ações do programa em uma lista. Essa escolha foi utilizada para tornar o código mais fácil de ser entendido. A Figura 8.2 mostra um exemplo com as informações do celular.

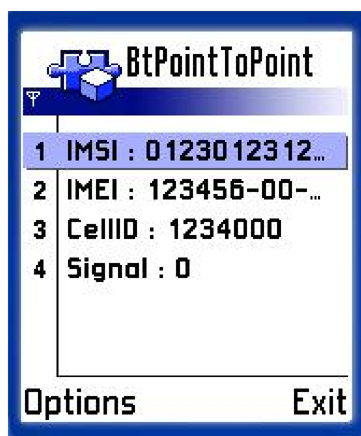

Figura 8.2 Exemplo de interface.

A implementação dessa interface é baseada em uma interface chamada MLog. As funções que devem ser implementadas são mostradas em Listagem 8.1. Elas funcionam de forma similar, todas escrevem uma mensagem na tela, a diferença é o formato da entrada. O objeto que implementa na aplicação é CBTPointToPointAppView.

Listagem 8.1 Funções da interface MLog

```
virtual void LogL(const TDesC& aText) = 0;
virtual void LogL(const TDesC& aText, const TDesC& aExtraText) = 0;
virtual void LogL(const TDesC& aText, TInt aNumber) = 0;
```

8.2 MENU

O menu tem sete itens. Como mostrado na seção 4.6, a definição do menu é feita no arquivo .rss. O trecho definindo o menu é mostrado em Listagem 8.2.

Listagem 8.2 Definição do menu no arquivo .rss

```
RESOURCE MENU_PANE r_btpointtopoint_menu
    {
    items =
        {
        MENU_ITEM {command = EBTPointToPointConnect;  txt = "Conecta ao GPS";},
        MENU_ITEM {command = EBTGetGPSData;  txt = "Obtem Dados GPS";},
        MENU_ITEM {command = EBTShowPOS;  txt = "Mostra Pos";},
        MENU_ITEM {command = EBTShowCellINFO;  txt = "Mostra RBase";},
        MENU_ITEM {command = EBTShowMobInfo;  txt = "Mostra MobInfo";},
        MENU_ITEM {command = EBTPointToPointClearList;  txt = "Limpa Tela";},
        MENU_ITEM {command = EAknSoftkeyExit;   txt = "Exit";}
        };
    }
```

Para facilitar o uso do programa, alguns itens do menu só devem aparecem sobre determinadas condições. Usando a função DynInitMenuPaneL, explicada na seção 4.6.3, os itens EBTPointToPointConnect, EBTGetGPSData, EBTShowPOS são manipulados conforme a situação da conexão com o GPS. A implementação é mostrada Listagem 8.3.

Listagem 8.3 Implementação do DynInitMenuPaneL

```
void CBTPointToPointAppUi::DynInitMenuPaneL(TInt aResourceId, CEikMenuPane* 
aMenuPane)
    {
    if (aResourceId == R_BTPOINTTOPOINT_MENU)
        {
  // GPS não está conectado
        if (!iClient->IsConnected())
            {
            aMenuPane->SetItemDimmed(EBTPointToPointConnect, EFalse);
            aMenuPane->SetItemDimmed(EBTShowPOS, ETrue);
            aMenuPane->SetItemDimmed(EBTGetGPSData, ETrue);

            }
// GPS está conectado

        else if (iClient->IsConnected())          {
             aMenuPane->SetItemDimmed(EBTPointToPointConnect, ETrue);
```

```
        aMenuPane->SetItemDimmed(EBTGetGPSData, EFalse);
        // Informações do GPS porcessadas
        if(iClient->isParsed){
                aMenuPane->SetItemDimmed(EBTShowPOS, EFalse);


        }else{
                aMenuPane->SetItemDimmed(EBTShowPOS, ETrue);

        }

       }

     aMenuPane->SetItemDimmed(EBTPointToPointClearList, !iAppView-
>ContainsEntries());
     }
  }
```

A situação inicial do menu no emulador é mostrada na Figura 8.3.

Figura 8.3 Menu Inicial.

8.3 FUNCIONALIDADES

Abaixo serão mostrados com foi implementado as três funcionalidades descrita no começo do capítulo. Devido a limitações do emulador, as telas mostradas serão do celular Nokia 6600.

8.3.1 Obter os dados GPS via bluetooth

Dentro as três funcionalidades, essa é a mais complexa. O primeiro passo é verificar se o GPSSlim236 [26] está funcionado e conectado a rede GPS. Isso acontece quando o LED laranja e o LED azul estão piscando. Agora, o item do menu “Conecta ao GPS” deve ser selecionado. Uma busca por dispositivos bluetooth é iniciada. O resultado esperado é parecido com o mostrado na Figura 8.4. Deve-se escolher o item “HOLUX GPSSlim236”. Após escolhido o dispositivo, a interface mostra os passos da conexão com mostrado na Figura 8.5.

Figura 8.4 Tela de busca de dispositivos.

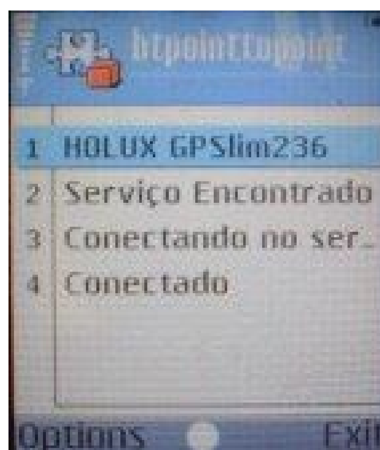

Figura 8.5 Conectado ao GPSSlim236.

Nesse estado de conectado, o item de obter as informações do GPS fica ativo e a opção de se conectar ao GPS fica desativada. Isso é detectado pelo resultado do retorno da função IsConnected() do objeto iClient. Esse novo menu é mostrado em Figura 8.6

Figura 8.6 Novo Menu com o GPS conectado.

Selecionando a opção "Obtem Dados GPS", o *socket* é lido com os dados enviado pelo GPS no formato NMEA (veja 2.2.2.1). Esses dados são manipulados para criar uma estrutura de dados com os valores de longitude e latitude atual. Quando isso for preenchido com dados válidos, um novo item no menu aparece denominado "Mostra Pos". Quando é selecionado, a interface apresenta na tela, os dados dessa estrutura como mostra a Figura 8.7.

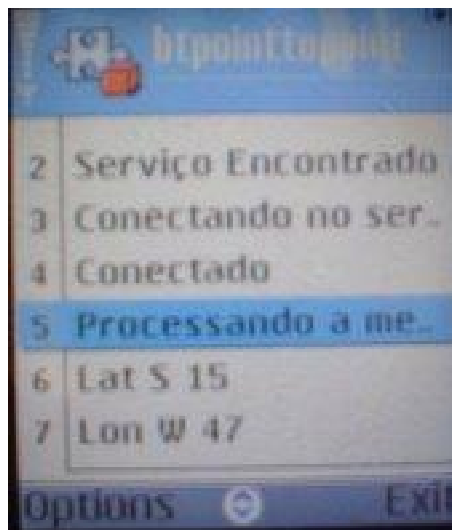

Figura 8.7 Dados do GPS.

8.3.2 Obter dados do telefone via MobInfo

Detalhes da biblioteca MobInfo foram mostrados na seção 7.2. Os dados reaquistados no mobinfo são: IMSI,IMEI,*Cell ID* e a potencia do sinal recebido. Pela documentação do MobInfo [4] , a potencia do sinal tem cinco níveis, sendo o cinco o mais forte. Um exemplo de resultado na tela do celular é mostrado na Figura 8.8.

Figura 8.8 Resultado do MobInfo

8.3.3 Obter dados do telefone via RBasePhone

De forma similar da seção 8.3.2, os dados são obtidos pelo método descrito em 7.1. O resultado no celular é mostrado em

Figura 8.9 Resultado do RBasePhone.

9 CONCLUSÃO

Durante o desenvolvimento desse projeto, foi descrito processo de desenvolvimento de uma aplicação baseada na localização detalhada no capítulo 8. Isso envolve explicar desde os elementos básicos dessa linguagem aos tópicos avançados como a comunicação bluetooth para obter os dados do hardware GPS externo.

9.1 RESULTADOS

Algumas limitações do MIDP 1.0 apresentada no item 1.1 poder ser superadas utilizando-se o Symbian C++. O capítulo 6 mostra como implentar a comunicação via bluetooth. O capítulo 7 mostrou como obter informações sobre o dispositivo e a rede GSM.

9.2 DIFICICULDADES SOBRE DESENVOLVIMENTO SYMBIAN

A linguagem Symbian C++ tem muitas peculiaridades mesmo para um experiente desenvolvedor C++. Alguns exemplos são a implementação de multitarefa por objetos ativos, a forma de tratar os erros pelo `trapd()`. Além disso, existe pouca documentação sobre o desenvolvimento dessa linguagem. As ferramentas abertas ainda estão em estágios inicias. Apesar dessas dificuldades, existem muitos programas feitos para Symbian. Vários foram migrados do *Palm OS* para essa linguagem.

Outro problema foi que devido a incompatibilidade da biblioteca MobInfo com a versão 6.1 do Symbian, a IDE de desenvolvimento foi trocada do Borland Builder para o Carbide C++, isso possibilitou comparar o uso entre essas duas IDEs. A Carbide C++ ainda está em fase inicial e a importação de projeto ainda é falha e necessita de configuração manual para criar o executável para o celular. As bibliotecas devem ser incluídas no perfil de simulação e no perfil para o celular, isso acarretou um trabalho dobrado para incluir uma nova biblioteca ao projeto. Além de, gerar erros quando os perfis não estavam com as bibliotecas iguais. Apesar desses defeitos, a interface do Carbide C++ é mais fácil para manipular os arquivos e trocar o binário de simulação para o de produção.

Um fato que atrasou o desenvolvimento foi as diferenças entre o emulador e o dispositivo real. As funcionalidades dos Capítulos 6 e 7 não estávam disponíveis no emulador por certas razões. A comunicação bluetooth do emulador somente funciona

utilizando duas portas RS-232 conectadas entre si e atualmente é muito raro encontrar um computador que tenha isso disponível. As informações sobre o celular fornecidas pelo emulador não podem ser configuradas, sempre geram um mesmo conjunto de valores de testes, isso limita muito os testes possíveis dentro do emulador.

9.3 MELHORIAS E NOVOS PROJETOS

Durante o desenvolvimento desse trabalho, descobriu-se que é possível acessar bibliotecas nativas em Symbian dentro da *virtual machine*. Isso permita que grande parte dos recursos do Symbian possam ser integrados ao um programa Java. Exemplo dessa solução é mostrado em [25]. Isso não invalida o trabalho desenvolvido nesse projeto pois as bibliotecas ainda devem ser feitas em C++, mas a interface poderia ser feita em Java por ser mais simples.

Uma grande aplicação de localização poderia ser desenvolvida com os conhecimentos apresentados. Uma sugestão seria migrar o trabalho desenvolvido em Java na referencia [9] para a linguagem Symbian. Sendo assim, esse trabalho ganharia em velocidade e novos recursos poderiam ser agregados como validar o usuário pelo IMEI ou IMSI e utilizar o *cellid* como outra forma de localização dentro do programa.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] EDWARDS,Leigh,et al. Developing Series 60 Applications. Addison Wesley, 2004

[2] HARRISON,Richard. Symbian OS C++ for Mobile phones. Wiley, 2004

[3] SYMBIAN. Symbian Developer Library v7.0s. Acessado em 05/06/06. Disponível em: http://www.symbian.com/developer/techlib/v70sdocs/doc_source/index.html

[4] SYMBIAN. Mobinfo 3rd Party Telephony Library. Acessado em 04/05/06. Disponivel em : http://www.symbian.com/developer/downloads/syslibs.html

[5] RAENTO,Mika. Getting the current Cell Id. Acessado em 03/08/05. Disponivel em : http://www.cs.helsinki.fi/u/mraento/symbian/cellid.html

[6] EDWARDS,Leigh,et al. Developing Series 60 Applications Code Samples. Acesado em 07/03/05. Disponivel em : http://www.forum.nokia.com/main/1,,40_2,00.html#code

[7] NOKIA. Carbide.c++ Express. Disponivel em: http://www.forum.nokia.com/main/0,,034-1001,00.html

[8] FRENCH, Gregory. Understanding The Gps - An Introduction To The Global Positioning System. GeoResearch,1996

[9] LIMA, Bruno; BORGES, Carlos; OLIVEIRA, Daniel; SOUZA, Wanessa. Sistema de Tomada de Decisão Baseada em Localização. Projeto Final de Graduação,Departamento de Engenharia Elétrica, Universidade de Brasília , Brasília , DF.

[10] STEELE,Raymond; LEE,Chin-Chun e GOULD Peter. GSM, cdmaOne and 3G Systems Wiley 2001

[11] BENNETT, Peter. The NMEA FAQ. Disponivel em: http://vancouver-webpages.com/peter/nmeafaq.txt

[12] NMEA, NMEA 0183 standard for interfacing marine electronic devices. Disponivel em : http://www.nmea.org/pub/0183/index.html

[13] WIKIPEDIA. UUID. Acessado em 06/03/2006. Disponível em : http://en.wikipedia.org/wiki/UUID

[14] SYMBIAN, Bluetooth Sockets, Disponivel em :
http://www.symbian.com/developer/techlib/v70sdocs/doc_source/reference/cpp/BluetoothSockets/index.html

[15] SYMBIAN, Bluetooth Service Discovery Database, Disponivel em :
http://www.symbian.com/developer/techlib/v70sdocs/doc_source/reference/cpp/BluetoothServiceDiscoveryDatabase/index.html

[16] BLUETOOTH, Specification Documents, Disponivel em :
http://www.bluetooth.com/Bluetooth/Learn/Technology/Specifications/

[17] ALAJÄRVI, Petri, SysInfo for Series 60, Disponivel em :
http://www.newlc.com/SysInfo-for-Series-60.html

[18] SYMBIAN, System Documentation, Disponivel em :
http://www.symbian.com/developer/techlib/index.asp

[19] ECLIPSE FOUNDATION, Eclipse IDE, Disponivel em :
http://www.eclipse.org/

[20] ACTIVE State, Active Pearl, Disponivel em :
ftp://ftp.activestate.com/ActivePerl/Windows/5.6/ActivePerl-5.6.1.635-MSWin32-x86.msi

[21] NOKIA, Series 60 SDK 2nd Edition, Disponivel em :
http://sw.nokia.com/id/13723492-3d79-460c-9764-727833570b8c/s60_sdk_v2_0_CW.zip

[22] SUN, Java 2 Platform Standard Edition 5.0 JRE, Disponivel em :
http://java.sun.com/j2se/1.5.0/download.jsp

[23] SYMBIAN, RSocket, Disponivel em :
http://www.symbian.com/developer/techlib/v70sdocs/doc_source/reference/cpp/SocketsClient/RSocketClass.html#%3a%3aRSocket

[24] Prasad, Maneesh, Location based services, Disponivel em:
http://www.gisdevelopment.net/technology/lbs/techlbs003.htm

[25] MIDP JNI, Using Symbian DLL in J2ME, Disponivel em:
http://www.midpjni.com/

[26] HOLUX, GPSlim236 Bluetooth GPS Receiver, Disponivel em:
http://www.holux.com/product/search.htm?filename=gpsreceiver_bluetooth_gpslim236.htm&target=gpsreceiver00&level=grandsonson

[27] NOKIA, Nokia 6600 Device Specifications, Disponivel em :
http://www.forum.nokia.com/main/0,,018-2209,00.html?model=6600

[28] LEE Wei Meng, Personalized Services Using Location Based Services (LBS), Disponivel em : http://conferences.oreillynet.com/cs/et2004/view/e_sess/4638

[29] SNAPTRACK, Location Technologies for GSM, GPRS and UMTS Networks, Diponivel em http://www.cdmatech.com/download_library/pdf/location_tech_wp_1-03.pdf

[30] STALLINGS,William. Wireless Communications and Networks.Pearson Prentice Hall, 2003.

[31] SYMBIAN, História do Symbian, Disponível em http://www.symbian.com/about/overview/history/history.html

[32] INOVA MOBILE, Página Oficial, Disponível em: http://www.gat.ene.unb.br/lemon.html

APÊNDICES

APÊNDICE A - CD DE INSTALAÇÃO

O CD de instalação agrupa os arquivos com as ferramentas e os códigos desenvolvidos nesse projeto. A estrutura de diretórios é:

- Bibliotecas – Bibliotecas necessárias
    - MOBINFO – biblioteca para obter dados sobre o celular. Descrito o funcionamento na seção 7.2.
    - RBASICGSMPHONE – arquivos *header* para serem incluídos para utilizar o recurso descrito na seção 7.1.
- Exemplos – Códigos de exemplos descritos em mais detalhes em Apêndice B
- Ferramentas – Ferramentas para o desenvolvimento:
    - Borland – Ferramenta baseado no Borland Builder C++ descrito no Apêndice C
    - Carbide – Ferramenta baseado no Carbide C++ Express descrito no Apêndice C
    -
- Texto – O texto desse projeto em formato digital.

APÊNDICE B -EXEMPLOS

Os códigos de exemplos estão no CD de instalação descrito no Apêndice A.

Por ter sido usado duas IDEs, os exemplos estão divididos em dois diretórios.

O primeiro "Formato Borland" tem os exemplos compatíveis com o Borland Builder C++ que são:

- HelloWold – exemplo incial de interface gráfica. O Capitulo 4 usa ele como exemplo para ilustrar os conceitos.
- HelloText – primeiro exemplo do desenvolvimento symbian que é descrito pelo Capitulo 3.

O segundo diretório "Formato Carbide" tem os exemplos compatíveis com o Carbide C++ que são:

- BluetoothChat – exemplo de aplicação bluetooth que envia mensagens entre dois celulares. O Capitulo 6 usa esse exemplo como base para mostrar os conceitos.
- LbsCompleto – exemplo final descrito pelo capítulo 8.

APÊNDICE C - AMBIENTE DE DESENVOLVIMENTO

Antes de explicar a linguagem, a configuração do ambiente para o desenvolvimento vai ser mostrada. Alguns conceitos podem não ficar claros nesse tópico, mas nos próximos capítulos serão mostrados com mais detalhes. O foco agora é mostrar as ferramentas para desenvolver em Symbian C++. Existem algumas plataformas que rodam Symbian. Por exemplo, Série 60, 80, 90 e UIQ. Dentre estas, a plataforma série 60 é a que tem mais fabricantes, além de ser a que tem maior número de mais aparelhos lançados. Por esses motivos foi a plataforma escolhida para o desenvolvimento. A utilização de uma IDE facilita o desenvolvimento, principalmente na fase de debug. As IDE que se integram aos kits de desenvolvimento da Serie 60 são:

- Microsoft Visual C++
- Borland C++
- Metrowerks CodeWarrior C++
- Carbide.c++ Express

Apesar da versão mais recente do Symbian estar na versão 9, existem muitos celulares com versões mais antigas. Nesse projeto foram usadas duas versões de Symbian OS v6.1 no celular N-Gage e Symbian OS v7.0s no Nokia 6600. Para cada versão do Symbian foi utilizada uma ferramenta diferente.

Versão 6.1 – N-GAGE - Borland

Para essa versão, existe um conjunto de programas disponibilizado no fórum Nokia denominado **Borland C++ Mobile Edition for Series 60, Nokia Edition** que consistem em:

- Series 60 SDK 1.2 for Symbian OS
- a Perl environment
- Java™ 2 Runtime Environment (JRE),
- C++ Mobile edition plug-in from Borland
- C++Builder 6 Personal edition

Isso facilita muito pois a integração entre o emulador, cross-copliador e IDE . Durante a instalação, todas as configurações para integração entre as ferramentas do pacote são feitas automaticamente. Por isso, foi escolhido esse pacote para desenvolver os aplicativos.

O procedimento de instalação segue as seguintes etapas:

- Baixar da URL - http://www.forum.nokia.com/main/0,,034-49,00.html
- Descompactar e rodar o arquivo INSTALL\setup.exe
- Instalar todos os pacotes da lista

Após instalar, rode o Builder C++

Para iniciar o procedimento para criar um projeto do Symbian é necessário:

- Selecionar a opção **Other em File>New** como mostra a Figura 9.1

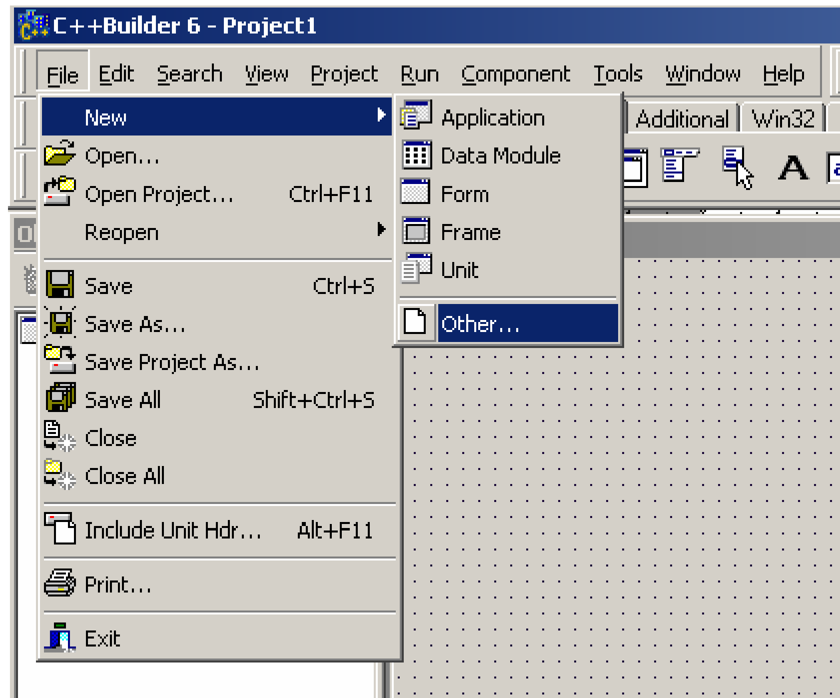

Figura 9.1 Criando um novo projeto

Escolhendo a aba de Mobile, existem duas opções como mostra a Figura 9.2. A opção de "*Import Mobile Application*" traz um código Symbian para a estrutura do Builder. Para isso deve utilizar o bld.inf do projeto como mostra a Figura 9.3. A outra é criar um novo projeto.

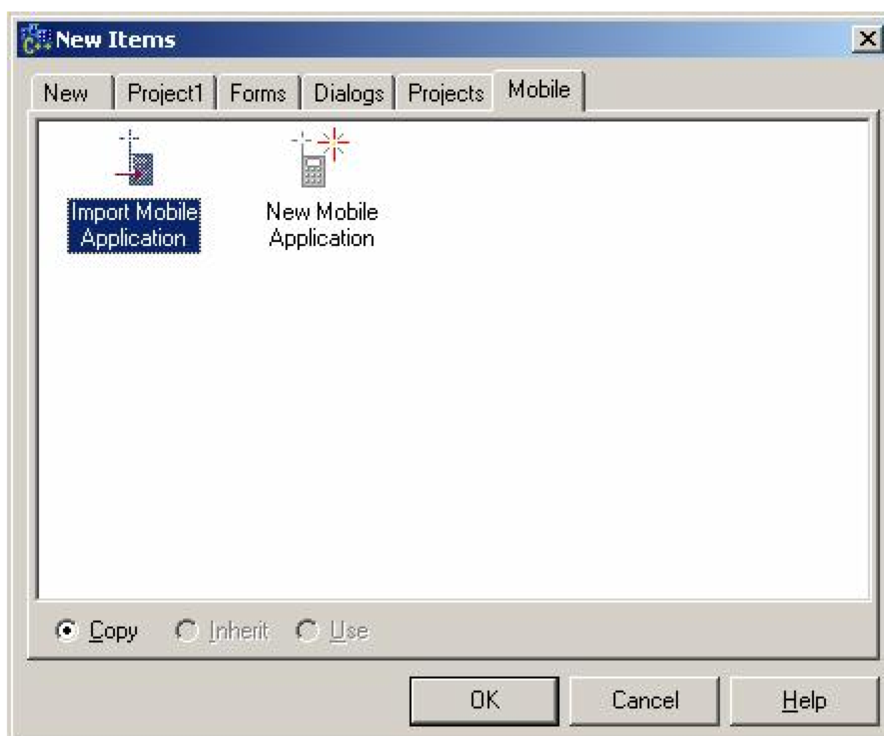

Figura 9.2 - Tipos do projeto móvel.

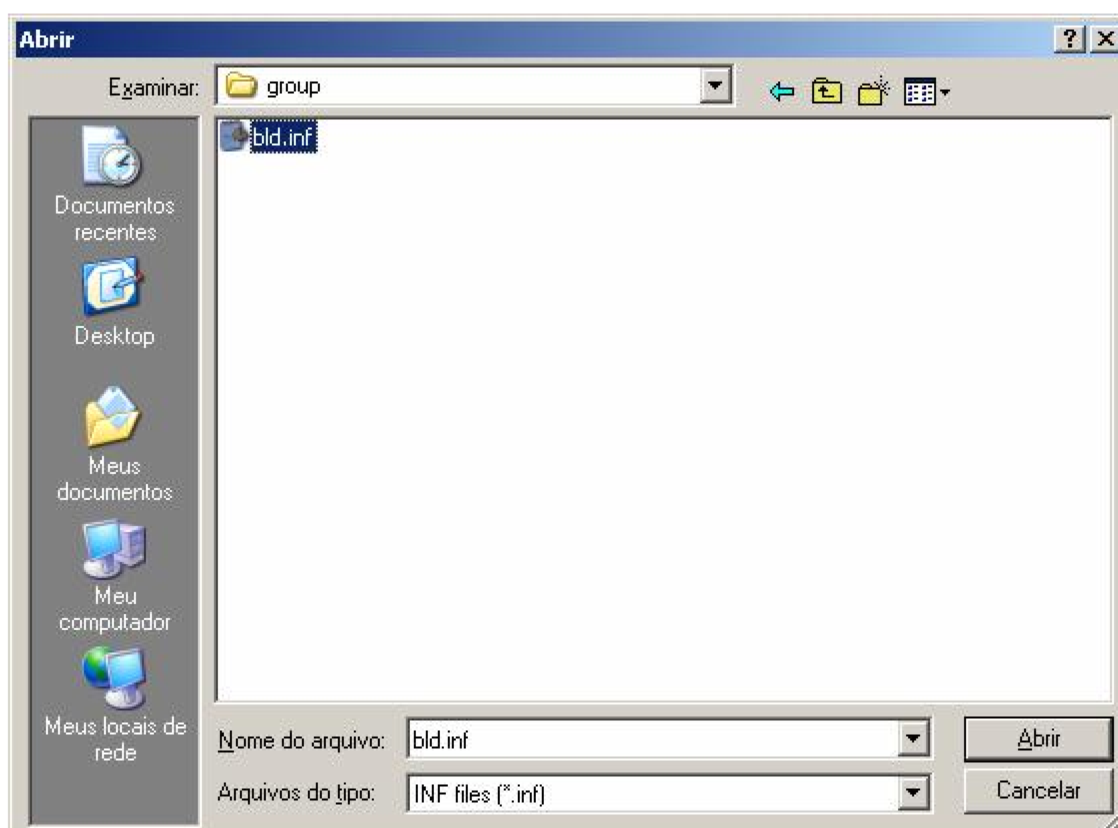

Figura 9.3 – Selecionar o arquivo bld.inf para importar um projeto.

Para criar um novo projeto é necessário mudar a opção na tela da Figura 9.2 e uma nova tela é mostrada ao usuário na Figura 9.4.

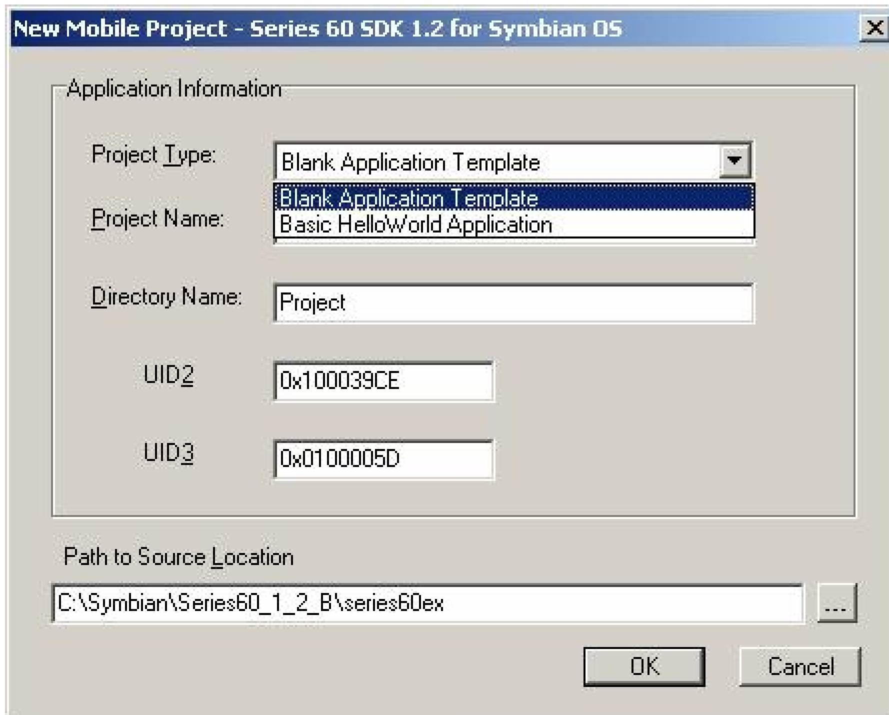

Figura 9.4 – Novo Projeto.

A primeira opção da Tela da Figura 9.4 é criar um projeto vazio ou criar um projeto funcional do tipo "*hello world*" muito simples. O UID2 é para definir o tipo de aplicação, o número 0x100039CE define uma aplicação com interface gráfica. O código UID3 é um número único para identificar a aplicação. Isso é explicado em mais detalhes em item 3.5

Versão 7.0s – NOKIA 6600 – CARBIDE C++

Entre as ferramentas mostradas no início deste capítulo, a Carbide.c++ Express é a mais nova. Ela foi criada pela Nokia para estudantes visando o desenvolvimento de aplicações não comerciais. A IDE é baseada no Eclipse versão 3.1 (version 3.1.1 exatamente) e Eclipse C/C++ *Development Tools Project*, CDT 3.0.1. [19]. Diferindo do pacote mostrado no item 0, em que todos os arquivos necessários estão em um único instalador, agora é necessário obter quatro programas diferentes para obter uma plataforma completa. Os programas são:

- Carbide C++ Express IDE [7]
- Active Pearl [20]

- S60 SDK segunda edição [21]
- Java Runtime Enviroment[22]

Java Runtime Enviroment

A primeira ferramenta a ser instalada é o programa para rodar arquivos Java. O

S60 SDK exige que uma versão maior que 1.3.1 esteja instalada. A versão utilizada é a 1.5.0 *update* 6. Na tela da Figura 9.5 basta selecionar a *Typical setup* e esperar a conclusão da instalação.

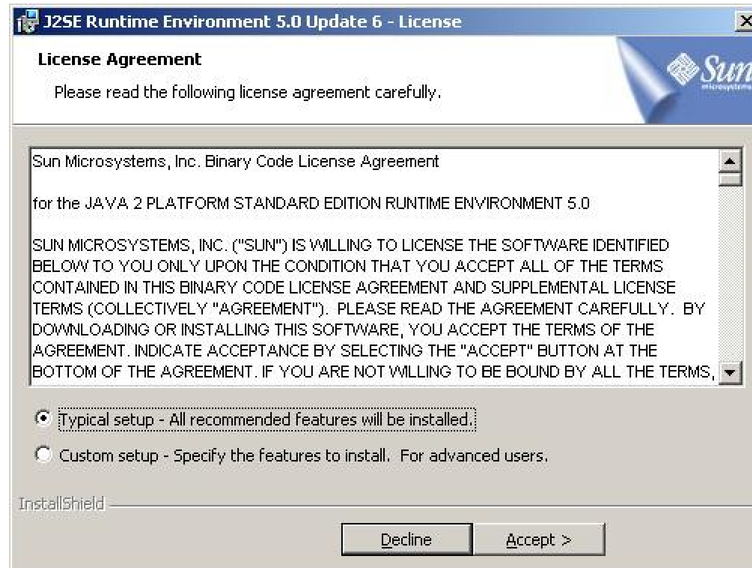

Figura 9.5 Instalação do JRE 1.5 update 6

Active Pearl

O Active Pearl é pré-requisito do S60 SDK. A exigência é uma versão maior que 5.18. A versão usada é 5.6.1.635.

Após algumas telas de contrato, surge a tela com os pacotes que devem ser instalados e todos os pacotes devem ser instalados como mostra a Figura 9.6.

Figura 9.6 Escolha de pacote no ActivePearl

Devem-se deixar as opções padrão nas telas posteriores. Outra tela em que todas as opções devem estar marcadas é mostrada na Figura 9.7. Com isso a instalação deve ocorrer com sucesso.

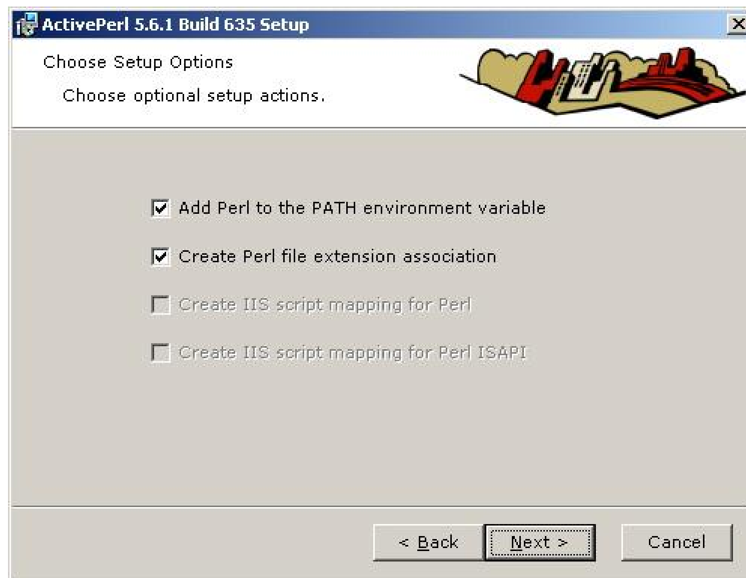

Figura 9.7 Opções do ActivePearl.

S60 SDK

A tela incial de instalação do S60 SDK pode gerar alguma confusão pois ele avisa os pré requisitos mas não detecta se eles estão instalados. Assim sendo, mesmo com os dois programas instalados corretamente, ele gera o aviso da Figura 9.8. O tipo de instalação que deve ser escolhido é o "*custom*" como mostra a Figura 9.9

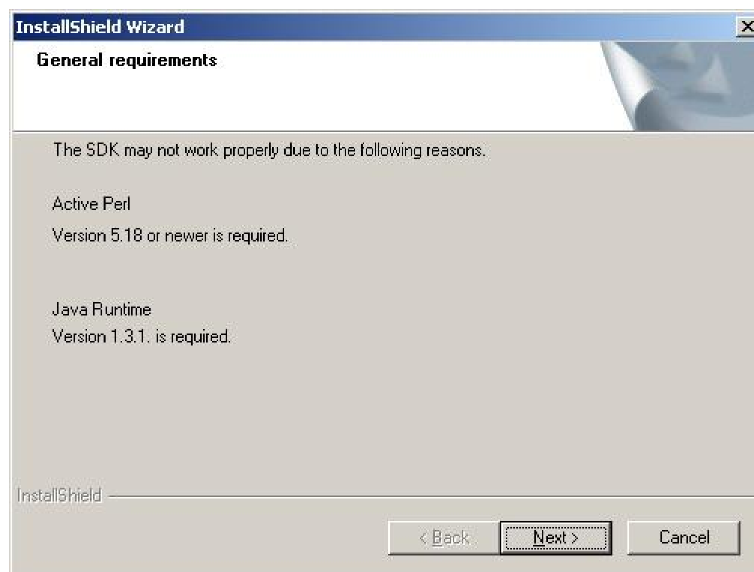

Figura 9.8 Aviso de pré-requisitos no S60 SDK.

Figura 9.9 Tipo de instalação do S60 SDK.

A próxima tela é o local onde vai ser instalado o S60 SDK. É recomendado o diretório padrão C:\Symbian\7.0s\Series60_v20_CW.

O próximo passo é escolher quais componentes devem ser instalados. O ideal é que tudo seja instalado como mostra a Figura 9.10.

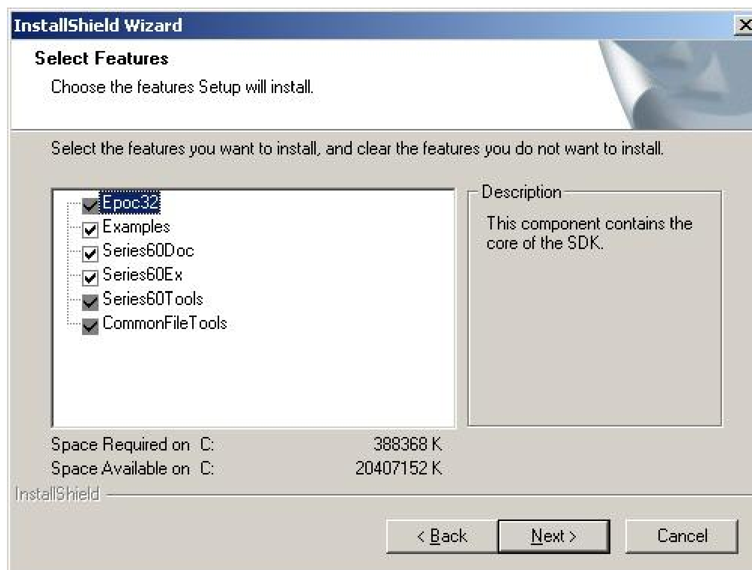

Figura 9.10 Instalação de pacotes do SDK.

Depois disso, é iniciado a instalação. O último passo é definir o SDK padrão. Se o programa da Borland estiver instalado, existirão duas opções. Nesse caso, pode manter a versão antiga como padrão. Isso não afeta o funcionamento do Carbide c++.

Carbide C++ Express

A instalação do Carbide é relativamente simples. A única pergunta relevante é se o diretório em que vai ser instalada a aplicação. O valor padrão é C:\Arquivos de programas\Carbide.

Na primeira execução do Carbide aparece uma caixa de diálogo mostrado na Figura 9.11, que pergunta qual o *workspace* deve ser usado e se o escolhido deve ser o padrão. Isso evita que seja perguntado novamente a cada nova execução.

Figura 9.11 Definição do *workspace*

A função do *workspace* é o espaço de trabalho isolado para cada usuário, que facilita o uso da IDE por várias pessoas sem interferir, uma na outra.

Assim como o Borland Builder mostrado no item 0, essa IDE pode criar um novo projeto ou importar um projeto.

Na criação de projeto, o primeiro passo é escolher o item “*S60 project*” como mostra a Figura 9.12. Depois, deve-se definir um nome para o projeto como mostra Figura 9.13.

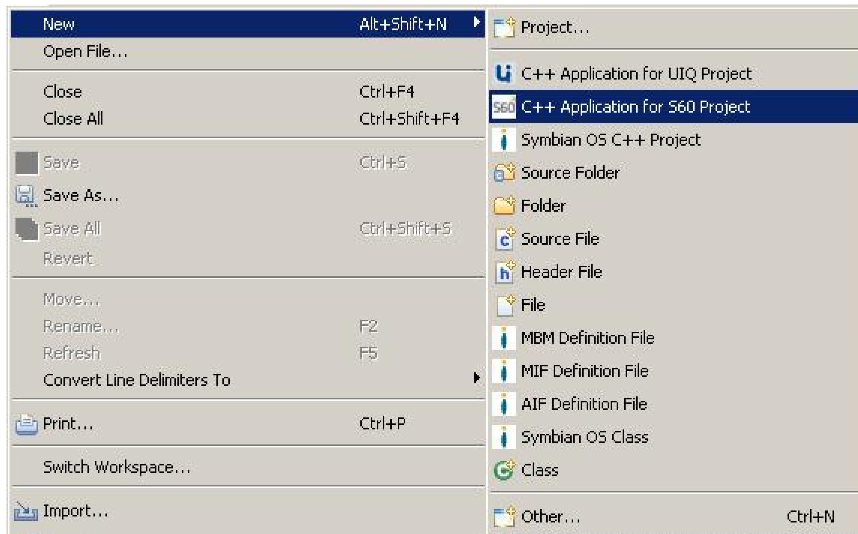

Figura 9.12 Criando um novo projeto.

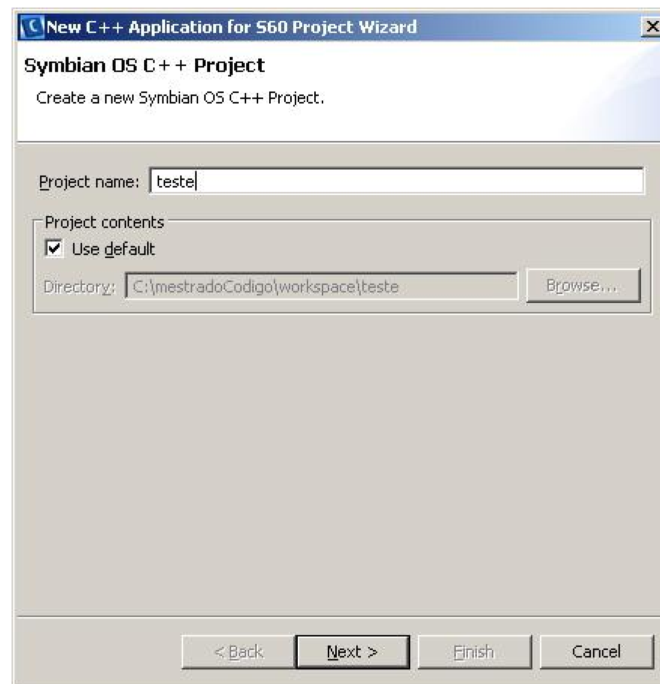

Figura 9.13 Nome do novo Projeto.

Existem várias opções de modelos de projeto. Cada um deles apresenta uma breve explicação e as versões de sdk são compatíveis como mostra a Figura 9.14. Por último, para criar o projeto deve-se selecionar qual SDK vai utilizar o projeto, como na □.

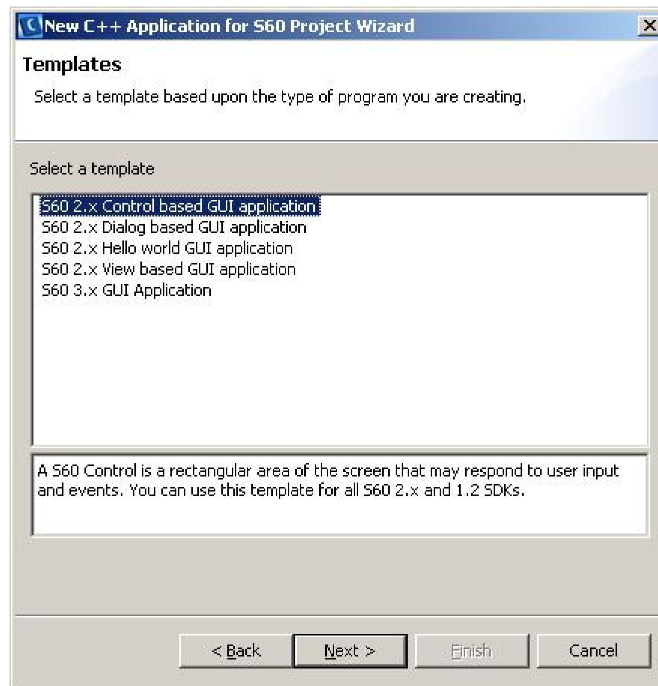

Figura 9.14 Tipos de Projetos.

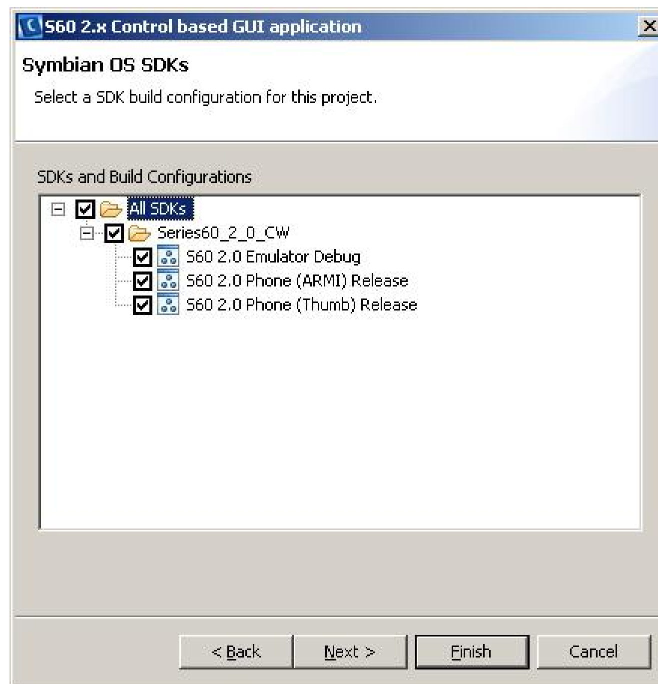

Figura 9.15 Escolha do SDK do projeto.

Outra opção é importar um projeto. Existem três opções para isso: (Figura 9.16)

- Importar um bld.inf

- Importar um arquivo .mmp
- Importar os executáveis

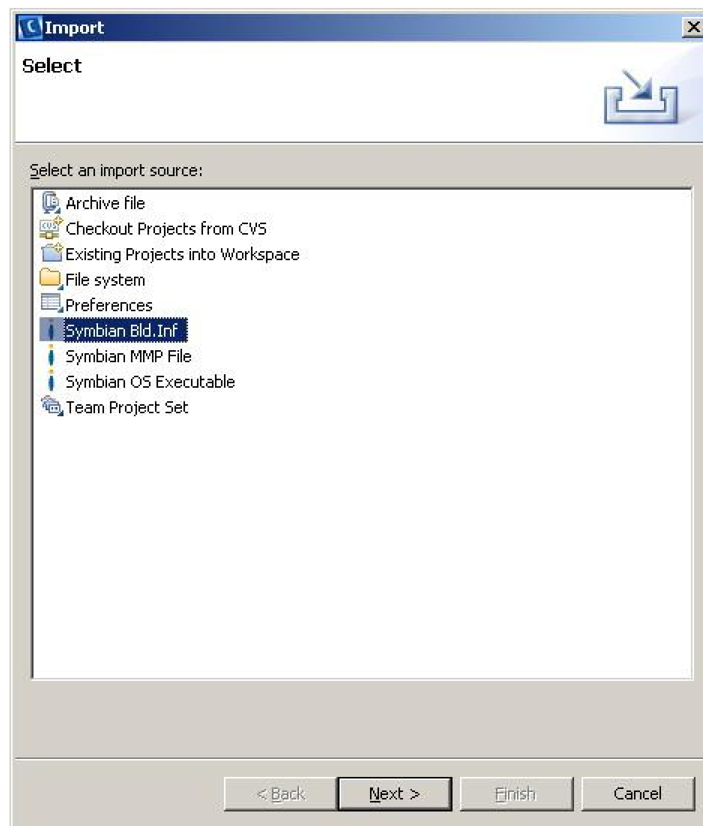

Figura 9.16 Importar projeto.

Os dois primeiros funcionam de forma análoga com a importação já vista no Builder. O item de importar executável é para executar um binário criado para o emulador.

Para efeito de comparação será importado um arquivo do tipo *bld.inf*.

A tela da Figura 9.17 é para escolher o arquivo *bld.inf* e optar se os arquivos associados serão copiados para o *workspace* ou serão usados no local original.

Figura 9.17 Escolhendo o bld.inf para importar.

Depois é feito um *parse* do arquivo .mmp apontado dentro do arquivo *bld.inf* na Figura 9.18

O último passo é escolher qual o SDK que vai associado ao novo projeto como é visto em Figura 9.19.

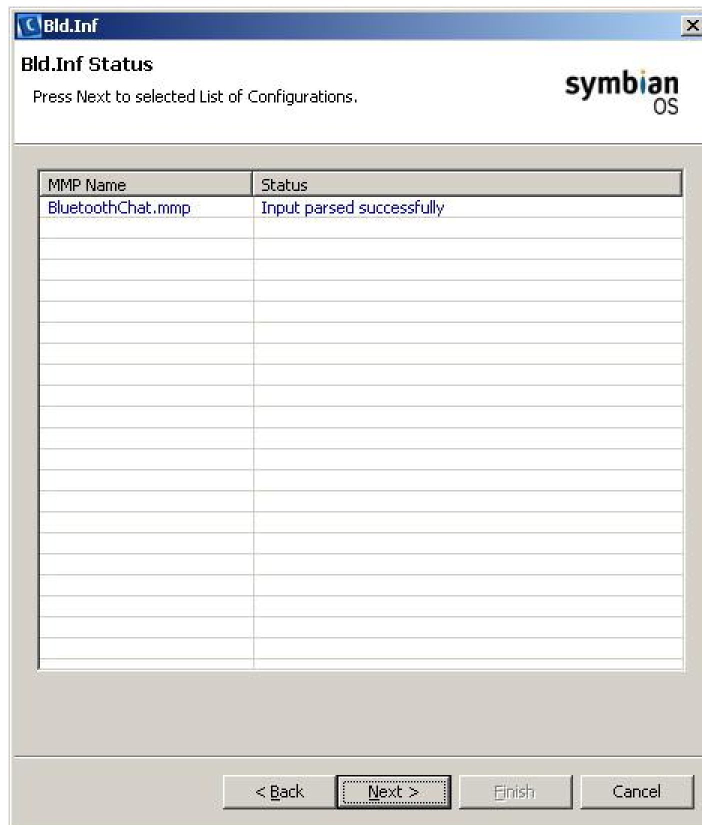

Figura 9.18 *Parse* do arquivo *.mmp*.

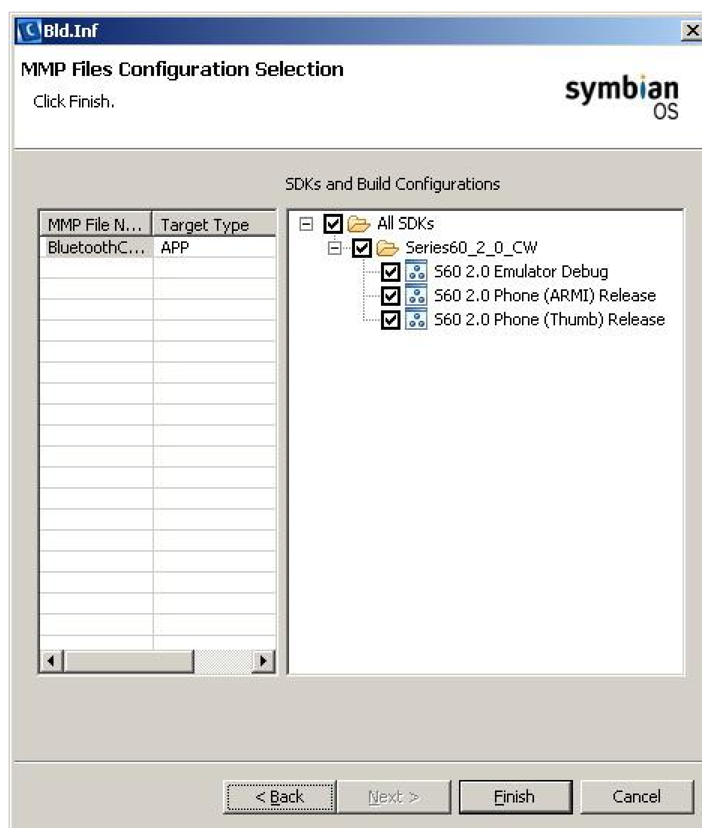

Figura 9.19 SDK usado pelo projeto importado.

Existe um problema na importação de projeto. O arquivo .pkg não é importado. Consequentemente, esse projeto não poderá ser usado para criar um arquivo .sis para rodar no celular.

Mas se um arquivo .pkg for criado dentro de um subdiretório /sis , esse problema será corrigido. Infelizmente esse processo é manual.