Symbian Platform Overview

Operating System?

- Software Program—Similar in this Sense to Other Programs
- Resource Encapsulation—Lens through which Users and Applications View System Resources like Hard disk, DVD drives, Networks.
Operating System?

- Binds Hardware and Software Together

Symbian OS—Overview

- Designed with Smartphones as the Target Platform in Mind
- Multitasking and Multithreading Supported
- Symbian OS Kernel is Microkernel
- Mobile Phone Manufacturers Buy Licenses of Symbian
Symbian OS—Layered Model

- Symbian OS is Structured in Layers
- Layers are Decomposed in Blocks and Sub-blocks
- Blocks and Sub-blocks are Decomposed in Components or Collection of Components
- Layers are Highest Level of Abstractions
- Components are Lowest Level of Abstractions
- Layers and Blocks are Logical Concepts
- Components are Physical Objects (Software Code)

Symbian OS—Layered Model

- Layers
  - Each layer abstracts the functionality of the layer beneath and provides services to the layer above
  - Examples
    - OS Services Layer
    - UI Framework Layer
- Blocks
  - A block or sub-block roughly corresponds to a “Technology Domain”
  - Examples:
    - Telephony Services
    - Network Services
- Components
  - Components are the basic entities of the model
  - Common, Optional and Replaceable functionality is defined at Component Level
Symbian OS—Layered Model

- Symbian OS is Shipped in Headless Configuration
  - Minimal User Interface
  - Not Production Quality User Interface
- Mobile Phone Manufacturers Either
  - Develop their Own Production Quality User Interface or
  - License a Suitable User Interface
- Production Quality User Interfaces Already Developed:
  - S60 (Series 60)—Developed and Licensed by Nokia
  - UIQ—Developed and Licensed by UIQ Technology
  - MOAP (Mobile Oriented Application Platform)—Developed by FOMA (Freedom of Mobile Access) Consortium in Japan
  - Series 80 and 90—Developed by Nokia but not Licensed to Others
Symbian Layers

- UI Framework Layer
- Application Services Layer
- OS Services Layer
- Base Services Layer
- Kernel Services & Hardware Interface Layer

(Book: Pg 14-15)

Symbian OS—Key Design Patterns

Localization Of Mobile Platforms : Pg 15
References

- The Symbian OS Architecture Sourcebook by Ben Morris
- Smartphone Operating System Concepts with Symbian OS by Michael J. Jipping

Symbian: Application Design and Architecture
(S60 Perspective)
Application Design: Typically MVC Pattern

- Model View Controller Architecture
  - Model (the Document—CEikDocument)
    - Contains and Manipulates Data of the Application
  - View (Application View—CCoeControl)
    - Displays Application State Based on Model Data
    - Receives User Input
    - Notifies Controller of Relevant Events
  - Controller (Application UI Controller—CEikAppUI)
    - Handles Application Events
    - Interacts with Model
    - Selects the View to be Displayed

Symbian Application Framework

- UIKON (Previously Called EIKON)
  - Main Component of “Application Framework”
  - Allows other GUI Frameworks to Run on Top of Symbian OS
    - S60
    - UIQ
- UIKON (Sub)Frameworks
  - CONE (Control Environment)
    - Framework for Graphical User Interface
  - APPARC (Application Architecture)
    - Framework for Applications and Application Data
S60 and UIQ Platforms

- Extend the UIKON Framework by Adding Libraries Appropriate for each of Them
- S60 Library: Avkon (Class Prefix: CAkn)
- UIQ Library: Qikon (Class Prefix: CQik)

Application Components

- Application
- Document
- AppUI
- View
Application Entry Point

• Every Symbian Application must Implement Two Functions that are Called by the Framework to Launch the Application

```cpp
LOCAL_C CApaApplication* NewApplication()
{
    return new CMyApplication;
}

GLDEF_C TInt E32Main()
{
    return EikStart::RunApplication(NewApplication);
}
```

Resources

• Resource Files are Used to Define:
  ◦ User Interface Components
  ◦ Visible Text
Resource Files

- Resource Files Contain:
  - GUI Element Definitions (Menus, Dialogs etc.)
  - Strings Needed by Application at Runtime
- Advantages
  - Make Source Code Shorter and Simpler
  - Save Memory because Text is Loaded only when Needed
  - Make Localization to Different Languages Easier

Resource File Structure

- Data Types
  - BYTE, WORD (2-bytes), LONG (4-bytes), DOUBLE (8-bytes)
  - LTEXT (A Unicode String with Defined Length)
  - BUF (A Unicode String)
  - LINK, LLINK (ID of Another Resource)
**Resource File Structure**

- **Resource File Statement Types**
  - **NAME**
  - **STRUCT**: Named Structure for Building Aggregate Resources
  - **RESOURCE**: Defines a Resource
  - **ENUM/enum**: Defines an Enumeration (Similar to C)

**STRUCT Statement**
- **STRUCT Statements are Placed in Files with .rh (Resource Header) Extension**

```c
STRUCT DIALOG {
    LONG flags=0;
    LTEXT title="";
    LLINK pages=0;
    LLINK buttons=0;
    STRUCT items[]; // an array
    LLINK form=0;
}
```
Resource File Structure

• RESOURCE Statement

```
RESOURCE DIALOG r messages_dialog
{
    title = "Title text";
    flags = EAnkDialogSelectionList;
    buttons = R_AVKON_SOFTKEYS_OK_CANCEL;
    items =
    {
        DLG_LINE
        {
            type = EAknCtlSingleListBox;
            id = EListControl;
            control = LISTBOX
            {
                flags = EAnkListBoxSelectionList;
                array_id = r_message_list;
            }
        }
    }
}
```

Resource Files: Bitmaps and Icons

• Application Icons are Stored in Bitmaps
• In Symbian, Multiple Bitmap Files are Stored in a Single File Called Multi Bitmap
• Bitmap Resources are Structure in Following Two Files
  ◦ MBM (Multi Bitmap File)
  ◦ MBG (Contains an ID for Each Bitmap in MBM)
Registration

- Applications are Required to be Registered with Underlying Platform
- File: `<application_name>_reg.rss`
  Contains Registration Information
  - UID of the Application
  - Name of Application Executable (without extension)
  - Application Properties (embedibility, hidden)

Localization

- Localization Files Contain “Local Language Strings”
- One Localization File is Produced for Each Language
- File: `<application_name>.rls`
  Contains Strings to be Localized
### Project Specification File (MMP)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TARGET</td>
<td>MyApp.exe</td>
</tr>
<tr>
<td>TARGETTYPE</td>
<td>exe</td>
</tr>
<tr>
<td>UID</td>
<td>0x0100039CE</td>
</tr>
<tr>
<td>SOURCEPATH</td>
<td>..\src</td>
</tr>
<tr>
<td>SOURCE</td>
<td>MyApplication.cpp</td>
</tr>
<tr>
<td>SOURCE</td>
<td>MyAppView.cpp</td>
</tr>
<tr>
<td>SOURCE</td>
<td>MyAppUi.cpp</td>
</tr>
<tr>
<td>SOURCE</td>
<td>MyDocument.cpp</td>
</tr>
<tr>
<td>SOURCEPATH</td>
<td>..\data</td>
</tr>
<tr>
<td>START RESOURCE</td>
<td>My.rss</td>
</tr>
<tr>
<td>RESOURCEHEADER</td>
<td>resource\apps</td>
</tr>
<tr>
<td>START RESOURCE</td>
<td>My_reg.rss</td>
</tr>
<tr>
<td>RESOURCEHEADER</td>
<td>private\0003a3f\apps</td>
</tr>
<tr>
<td>LANG</td>
<td>01</td>
</tr>
<tr>
<td>VENDORID</td>
<td>0</td>
</tr>
<tr>
<td>SECUREID</td>
<td>0xEA7408AF</td>
</tr>
<tr>
<td>CAPABILITY</td>
<td>ReadUserData</td>
</tr>
<tr>
<td>START BITMAP</td>
<td>MyApp.mbm</td>
</tr>
<tr>
<td>HEADER</td>
<td>Resource\Apps</td>
</tr>
<tr>
<td>SOURCEPATH</td>
<td>..\images</td>
</tr>
<tr>
<td>SOURCE</td>
<td>image1.bmp</td>
</tr>
<tr>
<td>SOURCE</td>
<td>image2.bmp</td>
</tr>
</tbody>
</table>

#### Project Specification File

- **UID Comprises 3 Components:**
  - **UID1:** Same for All Binary Files and Automatically Supplied
  - **UID2:** Indicates the Type of Executable (0x0100039CE for Applications)
  - **UID3:** Uniquely Identifies the Application

- **SECUREID**
  - By Default Same as UID3

- **CAPABILITY**
  - Specifying the APIs that Application want to Access
Application Resource Files

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppName.rss</td>
<td>Application’s Resource Script</td>
</tr>
<tr>
<td>AppName_reg.rss</td>
<td>Application’s Registration Information</td>
</tr>
<tr>
<td>AppName.rls or AppName.loc</td>
<td>Application’s Localizable Strings</td>
</tr>
<tr>
<td>AppName.rsg</td>
<td>Generated Header Containing Symbolic Resource IDs</td>
</tr>
<tr>
<td>AppName.hrh</td>
<td>Enumerated Constants for Application’s Commands</td>
</tr>
<tr>
<td>AppName.rsc</td>
<td>Generated Compiled Resource File</td>
</tr>
</tbody>
</table>

Application Architecture Possibilities
Traditional Symbian Control Based Architecture

- Localization of Mobile Platforms (Pg 45)
Dialog Based Architecture

- Localization of Mobile Platforms (Pg 45)
View Switching Architecture

References

- Symbian OS C++ for Mobile Phones by Richard Harrison and Mark Shackman
- S60 Programming by Paul Coulton and Reuben Edwards
- Developing Software for Symbian OS by Steve Babin
- The Accredited Symbian Developer Primer by Mark Jacobs and Jo Stichbury
- Mobile computing : technology, applications, and service creation by Asoke K. Talukder, Roopa R. Yavagal
Symbian: Application Development Concepts

Class Naming Conventions

- Class Names
  - Prefix+Class Name+Suffix

- C Classes (C is Prefix in Class Name)
  - Prefix 'C' Stands for 'Cleanup'
  - Derived Directly or Indirectly from CBase
  - Should be Constructed on Heap and Require Cleanup
  - Should Not be Constructed on Stack, Use Private/Protected Constructor to Prevent this
  - A Class can Only Inherit from a Single C Class
  - Example: CArray

- T Classes (T is Prefix in Class Name)
  - Also Called Data Type Classes
  - Encapsulates a Value of Specific Type e.g. TChar
  - Generally Do Not Use Dynamic Data i.e. Created on Stack but May also Use Heap if Required
Class Naming Conventions

- **R Classes (R is Prefix in Class Name)**
  - Also Called Resource (‘R’) Classes
  - Owns a Client Side Handle to a Resource, Resource is Actually Owned by a Symbian OS Server
  - Can be Instantiated on Heap or Stack
  - Example: RFile

- **M Classes (M is Prefix in Class Name)**
  - Also Called Interface Classes
  - Equivalent to an Abstract Class (Contains Pure Virtual Functions)
  - Used to Define Callback Interfaces
  - A Class can Inherit from Multiple M Classes

- **Static Classes (No Prefix Attached to Static Classes)**
  - Contains Only Static Functions
  - Cannot be Instantiated into an Object

Symbian Data Types *(e32defs.h)*

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Symbian OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>TInt, TInt64, TInt32, TInt16, TInt8</td>
</tr>
<tr>
<td>Unsigned</td>
<td>TUint, TUint32, TUint16, TUint8</td>
</tr>
<tr>
<td>Float</td>
<td>TReal, Treal64, Treal32</td>
</tr>
<tr>
<td>Character</td>
<td>TText, TText16, TText8, TChar</td>
</tr>
<tr>
<td>Boolean</td>
<td>TBool</td>
</tr>
<tr>
<td>void*</td>
<td>TAny*, (Can Point to a Function as Well)</td>
</tr>
</tbody>
</table>
Exception Handling

- Standard C++ Exception Handling in Symbian OS v9 (try/catch mechanism)
- Exceptions for Previous Symbian OS (before OS v9) are “Leaves”
  - Leaves are Alternative to C++ Exceptions

Exception Handling

- Exceptions or Leaves
  - Runtime Errors that are not Programmer’s Fault
  - Examples: Lack of Memory, Inability to Open Network Connection
- Panic
  - A Programming Error
  - Generally, an Application is Terminated in Case of Panic
  - Panics Cannot be Caught and Handled
  - Examples: Out of Bounds Array
Exception Handling: Leaves

- A Leave
  - Suspends Code Execution at the Point where Leave Occurs
  - Resumes Execution where Leave is “Trapped”
- Leaves May Occur while Performing Operations that May not Succeed
  - Allocation of Memory
  - File Creation
- Traps are used to Catch Leaves and Allow them to be Handled
- Functions that May Leave have “L” as Suffix in their Name

Why Do Leaves Occur?

- Calling a Leaving Function
- Use of Overloaded new (Eleave) Operator when Memory Allocation Fails
- Use of Explicit Leave [Similar to C++ Throw]
  - User::Leave()
  - User::LeavelfError(TInt)
  - User::LeaveNoMemory()
  - User::LeavelfNull()
How Leaves are Trapped?

- If a Function may Leave, it is Called Like This
  - TRAPD(error, FunctionMayLeaveL());
  - OR
  - Tint error;
  - TRAP(error, FunctionMayLeaveL());

Cleanup Stack

- Cleanup Stack is Crucial to Symbian Memory Management
- Used to Ensure that if a Leave Occurs, there are No Memory Leaks
- Cleanup Stack is Used to Store Pointers that may Become Orphaned if a Leave Occurs
Cleanup Stack Rules

- Any Locally Scoped Pointer to a Heap-Allocated Object must be Pushed onto the Cleanup Stack if there is a Risk of a Leave Occurring and there is no other Reference to the Object Elsewhere
- Instance Data (data owned by an instance of a class) Must Never be Pushed onto the Cleanup Stack

Cleanup Stack Functions

- To Push a Pointer on Cleanup Stack
  ◦ CleanupStack::PushL(aPointer)
- To Pop a Pointer from Cleanup Stack
  ◦ CleanupStack::Pop(aPointer)
- To Pop Multiple Items
  ◦ CleanupStack::Pop(aCount, aPointerToLastExpectedItem)
- To Pop and Destroy
  ◦ CleanupStack::PopAndDestroy(aCount, aPointerToLastExpectedItem)
2-Phase Object Construction

- Object Construction Steps on Heap
  - Step-1: Allocate Required Memory on Heap
  - Step-2: Execute Constructor of the Allocated Object
- What if Step-1 Succeed, and Step-2 Fails?
  - Allocated Memory will be Orphaned
- Solution
  - Perform Construction of Complex Objects in Two Phases

2-Phase Object Construction

- Make All Constructors Private or Protected
- Provide Static Factory Function(s) to Create Objects in Following Steps:
  - Allocate Memory on Heap Using Almost an Empty Constructor i.e. a Constructor that Cannot Leave
  - Push the Allocated Object Pointer on Cleanup Stack
  - Perform Construction of the Object Data
  - Pop the Allocated Object Pointer from Cleanup Stack
2-Phase Object Construction

- **Tips**
  - Constructors and Destructors Must Never Leave
  - Destructors Must Never Assume that Construction was Done in Full

2-Phase Object Construction Mechanism

- **new**: Almost Never Used. Used when Constructing a New Instance of an Application
- **new (ELeave)**: Used when Constructing a Heap-Allocated Instance of a Class
  - **NewL()**: Used when Constructing a Heap-Allocated Instance of a Compound Class [Cleanup from Cleanup Stack is Not Needed]
    - Instance is Either Directly Assigned to Member Pointer of Another Class Object OR
    - There is No Danger of a Leave before the Object is Deleted
  - **NewLC()**: Used when Constructing a Heap-Allocated Instance of a Compound Class [Cleanup from CleanupStack is Needed]
    - Instance is Assigned to a Locally Scoped Pointer and there is a Danger of a Leave before the Object is Deleted
References

- Mobile computing : technology, applications, and service creation by Asoke K. Talukder, Roopa R. Yavagal
- S60 Programming by Paul Coulton and Reuben Edwards
- Developing Software for Symbian OS by Steve Babin
- The Accredited Symbian Developer Primer by Mark Jacobs and Jo Stichbury
- Developing Series 60 Applications: A Guide for Symbian OS C++ Developers by Leigh Edwards
- http://www.symbian.com
- http://www.forum.nokia.com

Symbian: Application Development Concepts
Descriptors

- Descriptors in Symbian OS are Similar to Strings
- May Contain Text and Binary Data
- TPtr8—8 bit Characters—Narrow Descriptors
- TPtr16—16 bit Characters—Wide (Unicode) Descriptors
- Descriptors Can be
  - Constant (Contents are Constant)
  - Modifiable (Contents are Modifiable)

Descriptors

- Descriptors have an iLength Member that Stores the Current Length of the Descriptor
- Descriptors can Work without Null Termination
- Modifiable Descriptors also have an iMaxLength Member
- An Attempt to Increase the Length of the Descriptor beyond the Maximum Length will Result in an Immediate Panic
**Pointer Descriptors**

- Can point to text on the heap, or stack
- Constant: `TPtrC`

- Modifiable: `TPtr`

```
6  0x123456
```

**Stack Based Buffer Descriptors**

- Useful for relatively small size of data
- Directly contain data (as part of descriptor object)

- Constant: `TBuf<6>`

```
6  HELLO!
```

- Modifiable: `TBuf<8>`

```
6 8 HELLO!
```
Dynamic Descriptors (Heap Based)

- Can be Used for Strings
  - That are Too Big to be Placed on Stack
  - For Which Size is Not Known at Compile Time
- Constant: HBufC

```
const HBufC* buf = new HBufC(6, "HELLO!");
```

Dynamic Descriptors (Heap Based)

- Modifiable: RBuf

```
const RBuf* buf = new RBuf(6, 10, 0x234567);
```

```
buf->set(6, "HELLO!");
```
String Literals

- Literals are Strings, Generally Used for Printable Text in the Program
- Literals are of Type TLitC, TLitC8, TLitC16
- String Literals are Constructed using the _LIT Macro
  - _LIT(KText, “Hello World”);
  - Where KText is Name of String Literal (i.e. variable name)

Collection Classes: Arrays

- RArray
  - An Array of Fixed Length Objects
  - Size of One Array Element can Not Exceed 64o Bytes
- RPointerArray
  - An Array of Object Pointers
Collection Classes: Arrays

- CArray
  - Use Buffers to Store Data
  - Flat CArray
    - Store Entire Data in a Single Heap Cell
    - Once Full, any Append Operation Requires a New Heap Cell to be Allocated that is Large Enough to Contain the Original and New Data
  - Segmented CArray
    - Store Data in Doubly Linked List of Smaller Segments
    - Each Segment is a Separate Heap Cell of Fixed Size

Asynchronous Services

- All System Services are Provided through Servers
- Servers Operate in their Own Processes
- Service Provider APIs Typically have Asynchronous and Synchronous Versions of their Functions
- Service Request Function Returns Immediately, while the Request itself is Processed in the Background.
- Relevant Processes are Notified when Request is Complete
Asynchronous Services and Active Objects

- Symbian Allows Application Programs to Create Threads
- Multiple Asynchronous Services can be Accessed using Multiple Threads
- However, Symbian Recommendation is to Use Active Objects where Possible, as an Alternative Option

Active Object Framework

- Active Object Framework
  - Active Objects
  - Active Scheduler
- Active Object Framework is Used for Event Driven Multitasking
- Active Scheduler Maintains a List of Active Objects which have Made Request for an Asynchronous Service
Active Object

- Implement ‘Asynchronous Service Requesting Objects’ as an Active Object
- An Active Object:
  - Requests an Asynchronous Service and
  - Handles the Resulting Completion of Event Sometime After the Request.
  - May Ask to Cancel a Request
  - Is Listed with Active Scheduler

Active Scheduler

- When Asynchronous Service Completes, It Generates Events to Notify Active Scheduler
- Active Scheduler
  - Detects Service Completion Events
  - Determines Associated Active Object
  - Calls the Active Object to Handle the Event.
Submitting an Asynchronous Request

- Each Active Object Can Only Have One Outstanding Request
- If a Request has Already been Placed by an Active Object, a New Request may Result in
  - Panic
  - Refuse
  - Cancel Outstanding Request and Submit New One

Active Object Event Handling

- Active Object Implements the Event Handling Function
- Active Object Event Handler is the Function for Handling Completion of Asynchronous Call
- Active Object Handler is Not Pre-Empted
- Control Returns to the Active Scheduler When Event Handler Returns
- If Multiple Requests are Completed, Control Returns to the Scheduler, they are Handled Sequentially in Order of their Priority
- Active Scheduler Calls the Event Handling Function of Associated Object
Active Object Structure: Key Elements

- **iStatus**: Data Member Representing Request Status.
- **R-Class Object**: A Handle on the Asynchronous Service Provider (usually an R-class object).
- **Connection**: to the Asynchronous Service Provider.
- **Function**: to Issue the Asynchronous Request
- **RunL()**: Handler Function to be Invoked by the Active Scheduler when Request Completes
- **Cancel()**: Function to Cancel an Outstanding Request

Active Object Implementation

- Create a Class Derived from CActive
- Create Asynchronous Service Provider Handle (R-Classes) as a Data Member in the Class
- Invoke Constructor of CActive, Specifying Object Priority
- Connect to the Service Provider in ConstructL() Method
- Invoke CActiveScheduler::Add() in ConstructL()
- Implement NewL() and NewLC()
- Implement Asynchronous Request Function that Calls the Service, Specifying iStatus as the Argument.
- CallSetActive()
Active Object Implementation contd..

- Implement RunL() Method to Handle Necessary Work Once the Request is Complete
- Implement DoCancel() to Handle Request Cancel Operation.
- Implement RunError() to Handle any Leaves from RunL().
- Implement the Destructor to Call Cancel() and close the Handle(s) on the Service Provider(s).

Using Active Object

- Instantiate using NewL() or NewLC() as Appropriate

- Call Start(), to Make the Initial Request

- To Cancel the Request Prior to Completion, Call Cancel().
Active Objects: CActive Structure

- class CActive : public CBase {
  public:
  IMPORT_C virtual ~CActive();
  IMPORT_C void Cancel();
  inline TBool IsActive() const;
  inline TInt Priority() const;
  protected:
  IMPORT_C CActive(TInt aPriority);
  IMPORT_C voidSetActive();

  // Implements cancellation of outstanding request. This function is called as part of the active object's Cancel().
  virtual void DoCancel() =0;
  // Handles an active object's request completion event. A derived class must provide an implementation to handle completed request. The function is called by active scheduler when a request completion event occurs.
  virtual void RunL() =0;
  virtual TInt RunError(TInt aError); //Called by Active Scheduler if RunL() Leaves

  public:
  //Represents the status or error code returned by the asynchronous service provider.
  TRequestStatus iStatus;
}

Active Objects: Implemented CFileLoader

class CFileLoader : public CActive
{
  public:
  void Start();

  private:
  CFileLoader();
  void ConstructL(const TDesC& aFileName);
  void RunL();
  TInt RunError(TInt aError);
  void DoCancel();

  private:
  TFileName iFileName;
  RFile iFile;
};
ECOM

- ECOM is a Generic and Extensible Framework by which Abstract Interfaces can be Defined and their Implementations Identified, Loaded and Managed.
- ECOM is A Mechanism to Extend Symbian OS

What ECOM Does?

- **Identification**: Identifies all the Concrete Implementations of an Interface.
- **Resolution**: Allows the Client to Choose the Implementation to be Used
- **Instantiation**: Instantiates an Instance of the Concrete Class which Implements that Interface
ECOM (EPOC Component Object Model)

- ECOM Architecture is Used Transparently by Clients
- ECOM Server Manages Requests to Instantiate Concrete Instances of an Interface.
- ECOM Server Maintains a Registry of All Interface Implementations Installed on Device

ECOM Plug-in Interface Characteristics

- Standard Definition Functions
  - Abstract Class that Defines a Set of One or More Pure Virtual Functions
  - Concrete Classes Implement these Virtual Functions
- 1 or More Factory Functions
  - Used to Allow Clients to Instantiate an Interface Implementation Object
- Release Function
  - Used to Delete / Release Plugin
References

- Mobile computing: technology, applications, and service creation by Asoke K. Talukder, Roopa R. Yavagal
- S60 Programming by Paul Coulton and Reuben Edwards
- Developing Software for Symbian OS by Steve Babin
- The Accredited Symbian Developer Primer by Mark Jacobs and Jo Stichbury
- Developing Series 60 Applications: A Guide for Symbian OS C++ Developers by Leigh Edwards
- http://www.symbian.com
- http://www.forum.nokia.com