Symbian OS

UI-Development for S60

Disclaimer

- **These slides are provided free of charge at http://www.symbianresources.com and are used during Symbian OS courses at the University of Applied Sciences in Hagenberg, Austria (http://www.fh-hagenberg.at/)**
- **Respecting the copyright laws, you are allowed to use them:**
  - **for your own, personal, non-commercial use**
  - **in the academic environment**
- **In all other cases (e.g. for commercial training), please contact andreas.jakl@fh-hagenberg.at**
- The correctness of the contents of these materials cannot be guaranteed. Andreas Jakl is not liable for incorrect information or damage that may arise from using the materials.
- Parts of these materials are based on information from Symbian Press-books published by John Wiley & Sons, Ltd. This document contains copyright materials which are proprietary to Symbian, UIQ, Nokia and SonyEricsson. “S60™” is a trademark of Nokia. “UIQ™” is a trademark of UIQ Technology. Pictures of mobile phones or applications are copyright their respective manufacturers / developers. “Symbian ™”, “Symbian OS ™” and all other Symbian-based marks and logos are trademarks of Symbian Software Limited and are used under license. © Symbian Software Limited 2006.

Contents

- S60 UI Overview
- Resource Files and Localization
- Defining Menus
- Loading Strings from Resource Files
- Label Controls
- Modifying the Status Pane
- Dialogs
  - Overview
  - Custom Dialog
  - Notes
  - Query Dialogs

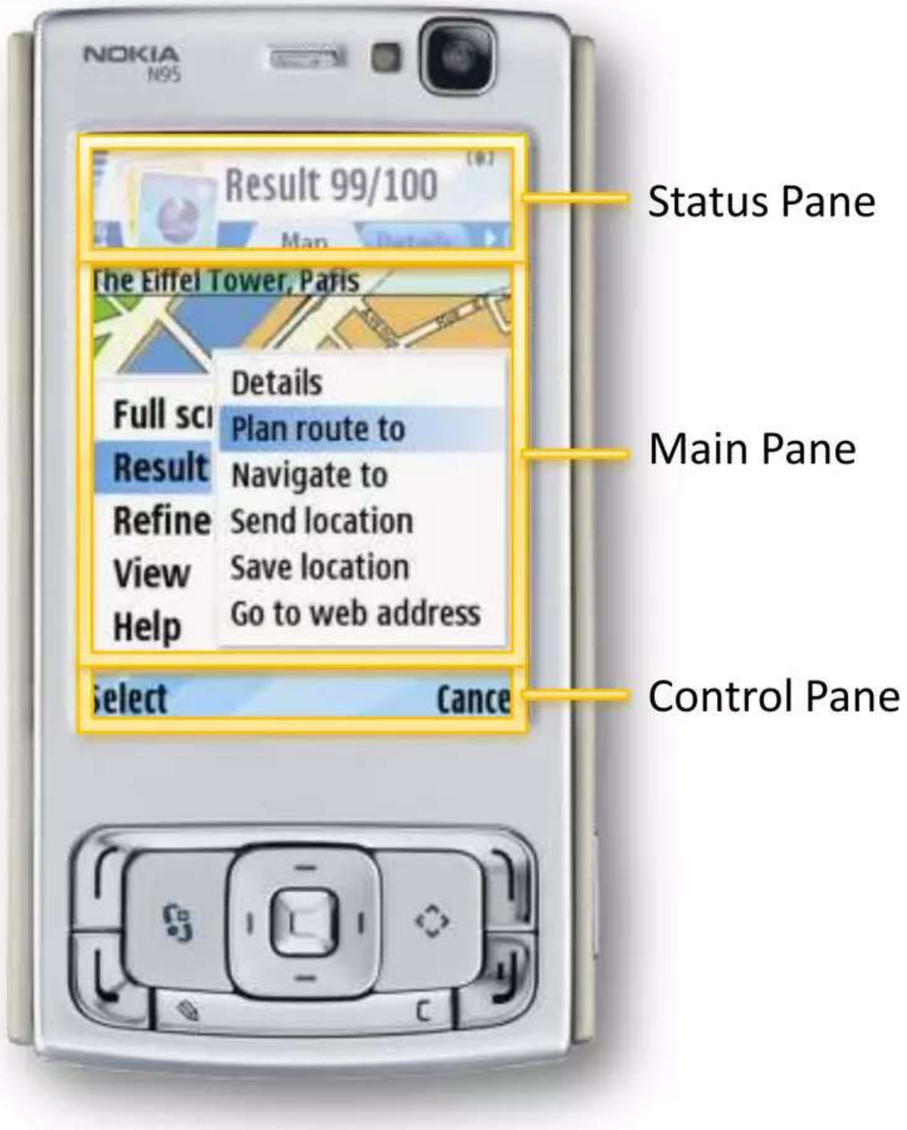
Elements of the

S60 UI

UI-Components

- **Window**
  - Sub-component of the device screen
  - Principal window filling up entire screen
  - Not used for display
  - Can contain many panes
- **Pane**
  - Sub-component of a window
  - Can contain many sub-panes

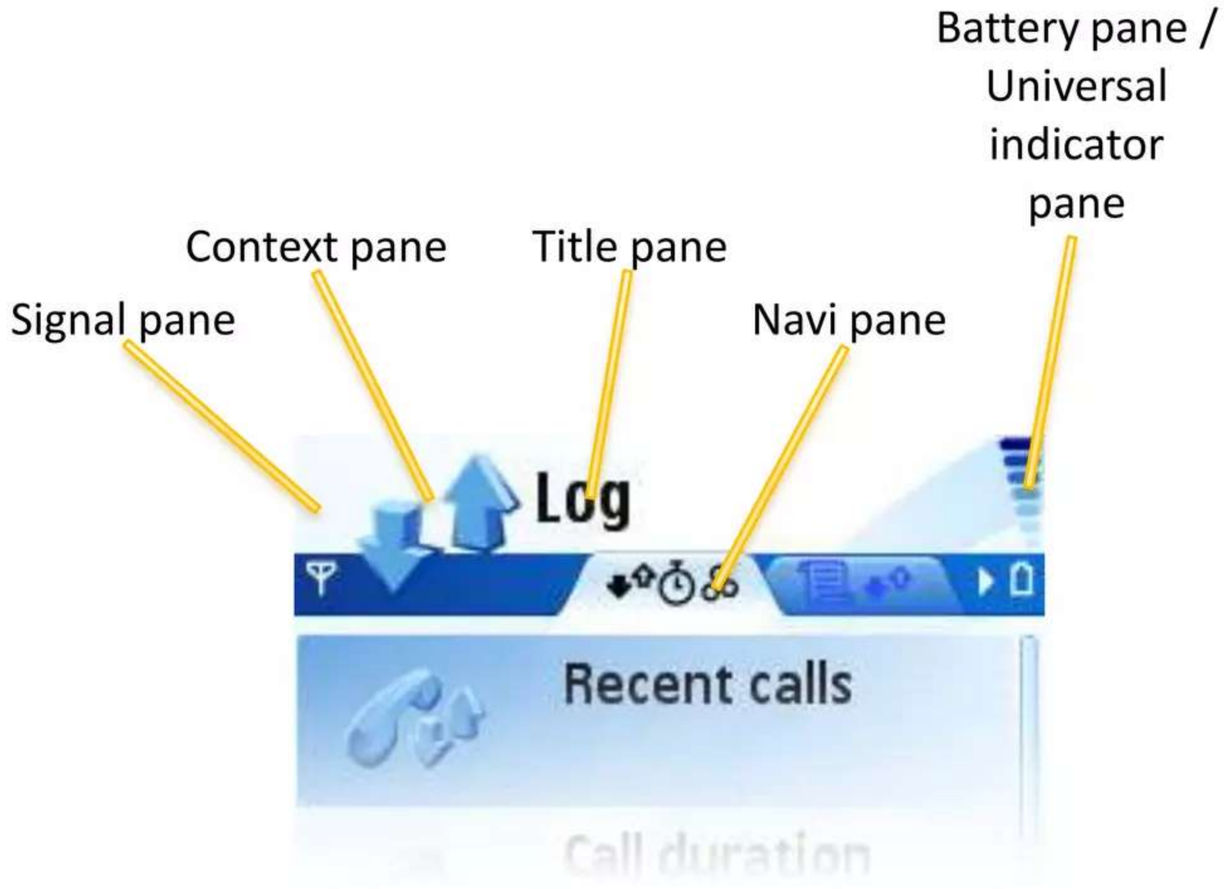
S60 UI



- **Status Pane**
  - Information about running application & device (e.g. battery strength)
- **Main Pane**
  - Application content
- **Control Pane**
  - Softkey labels

Status Pane

- Usually contains of sub-panes:
  - Title pane
  - Context pane
  - Navigation pane
  - Signal pane
  - Battery pane
- Automatically generated by AppUi



Navigation Pane

- Either empty or “decorated” with controls:
  - **Tabs**
    - Indicate different pages (= views)
  - **Navigation label**
    - Optional: Text on a tab
  - **Navigation image**
    - Optional: Image on a tab
  - **Indicators**
    - State of important element, e.g. volume
  - **Custom Controls**
    - Any CCoeControl-derived object

Define your UI

Resource Files

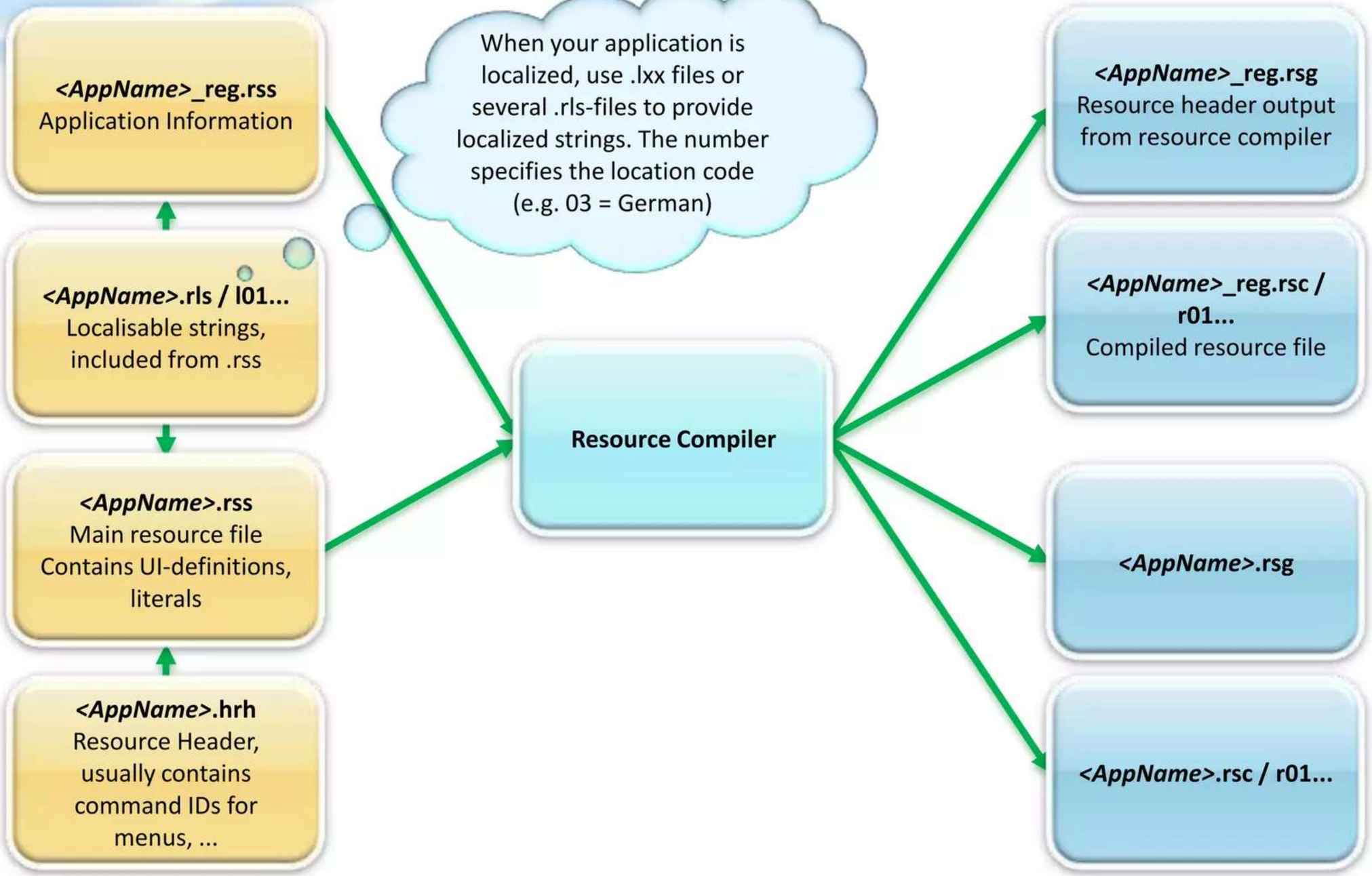
Resource Files

- Separate UI-elements from code
- **Define:**
  - User Interface (Menus, Dialogs, Lists)
  - (Localizable) Text
  - Application Information
- **Advantages:**
  - Information only loaded when needed (save RAM)
  - Localizable
  - Can be compressed

Applications & Resources

- App. has to have <*AppName*>.rss
- Can optionally use multiple resource files
- Resource files have their **own syntax**!
- **.rss-files compiled to .rsc + .rsg (= header)**
- **C++ code references** resources by including **.rsg-files**

Typical Logical Structure



File Type Overview

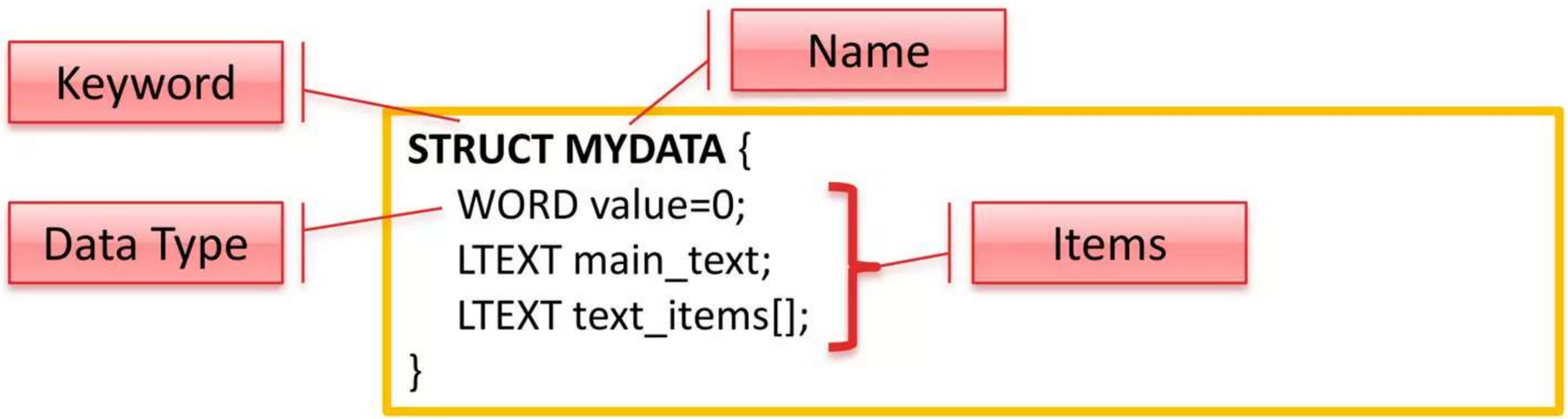
- Resource files and their meaning:

Type	Description
.rss	Plain text resource file, edited by the developer
.rh	Header-file used purely by a resource file
.hrh	Defines command IDs, shared by resource file & C++ Source
..._01.rls	Localization file for UK English (02 = French, 03 = German), Junction = usually directly in .rss. Defines Strings with *rls_string*-keyword.
.loc	Junction, includes currently compiled language definition file
.l01, ...	Localization file for UK English (02 = French, 03 = German). Defines Strings with *#define*-keyword
.rsc	Compiled resource file (independent resource compiler!)
.rsg	Header file for compiled resource file. Used by C++ source to reference elements in the .rsc-files.

Two different possible naming conventions

Resource File Format

- Consist of (multiple) **data constructs** with **uppercase keywords**
- **STRUCT keyword**
  - Sequence of items, each with name and data type
  - Not often needed directly



Resource File Format

- **RESOURCE-keyword**
  - Creates an **instance of a STRUCT**
  - **Assigns values** to its items
  - Predefined STRUCTs from Symbian OS / S60 / UIQ for UI-definitions, Strings, ...

```
// Creates an instance of STRUCT from the previous slide
RESOURCE MYDATA r_mydata_res {
    value=3;
    main_text="some text string";
    text_items={"item 1", "item 2"};
}
```

Resource File Structure

- Compulsory content of the main resource file:

HelloWorld.rss

```
NAME HELO                                   // 4 letter resource id

[ ... includes ... ]                        // Resource compiler handles includes like the C++ compiler
#include "DialogTest.hrh"                   // Defines enumerated constants for the application's commands
#include "DialogTest.rls"                   // (or .loc) – defines strings used in the UI.

RESOURCE RSS_SIGNATURE { }                  // optional version info

RESOURCE TBUF { buf = ""; }                 // can specify doc file – here: unused -> blank

RESOURCE EIK_APP_INFO r_application_dialog_test_app_ui {
   cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT;     // Softkey-definition (S60 predefined)
   status_pane = r_application_status_pane; // App-wide status pane (defined below in .rss)
}
```

STRUCTs are defined in eikon.rh

String Resources

- Define text in one place – and not in the source code
- Easy to modify, easy to localize

***<AppName>*.rls / .l01... (Text only!)**

```
rls_string STR_Loading “Loading...”
rls_string STR_Caption “Hello World”
```

***<AppName>*.rss (UI Definition)**

```
#include “<AppName>.rls”
RESOURCE LOCALISABLE_APP_INFO r_localisable_app_info {
   short_caption = STR_Caption; }
RESOURCE TBUF r_loading { buf = STR_Loading; }
[...]
```

***<Program>*.cpp (C++ Source Code)**

```
#include <stringloader.h>
#include <<AppName>.rsg>

void C<AppName>AppUi::DisplayInfo()
{
   HBufC* buf = StringLoader::LoadLC ( R_LOADING );
   [...]
   CleanupStack::PopAndDestroy(buf);
}
```

***<AppName>*.rsg (Generated by resource compiler)**

```
#define R_LOCALISABLE_APP_INFO          0x66a61005
#define R_LOADING                       0x66a61006
```

Localization

- Resource files compiled multiple times, once for each language
- Supported languages defined in .mmp

Journey.mmp

```
START RESOURCE Journey.rss
    HEADER
    TARGETPATH resource\apps
    LANG 01 03
END
```

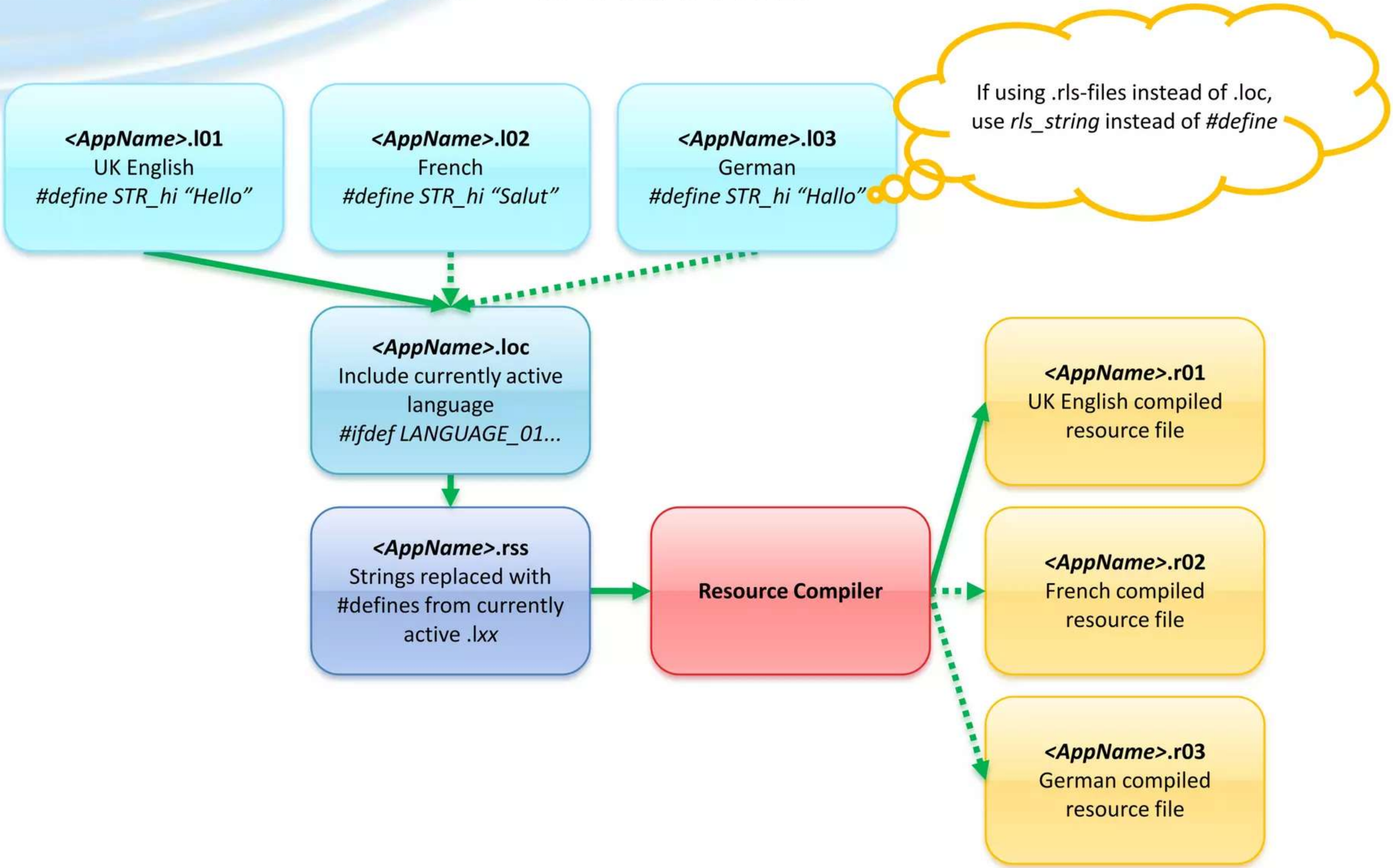
- Current language set by compiler using #define

Journey.loc

```
#ifdef LANGUAGE_01
        #include “journey.l01”        // UK English
#elif defined LANGUAGE_03
        #include “journey.l03”        // German
#else
        #include “journey.l01”        // Default to UK English
#endif
```

Two common naming conventions:
1.) .loc as junction & .l01... for each language
2.) .rls as junction & Journey_01.rls... for each language

Localization – Overview



Localized Installation

- Usually: **All languages contained in .sis-file**
- **Only language matching phone language is installed**
- .pkg-file defines which resource files to install for which language (all other files won't be installed)
- Installed resource-file renamed from .r01/.r*xx* to .rsc

Journey.pkg

```
&EN,GE                                          // Defines which languages are available
[...]
{
        "\Symbian\9.1\S60_3rd_MR\Epoc32\data\z\resource\apps\Journey.r01"
        "\Symbian\9.1\S60_3rd_MR\Epoc32\data\z\resource\apps\Journey.r03“
} -"!:\resource\apps\Journey.rsc“        // Only current phone language will get installed!
```

Internationalisation?

- … it's a lot more than translating text!

en_US	en_UK	de_DE	de_AT
MM/DD/YYYY	DD/MM/YYYY	DD.MM.YYYY	DD.MM.YYYY
5:24 PM	5:24 PM	17:24	17:24
$ 40.00	£ 40.00	40,00 €	€ 40,00
1,234.56	1,234.56	1.234,56	1.234,56
…	…	…	…

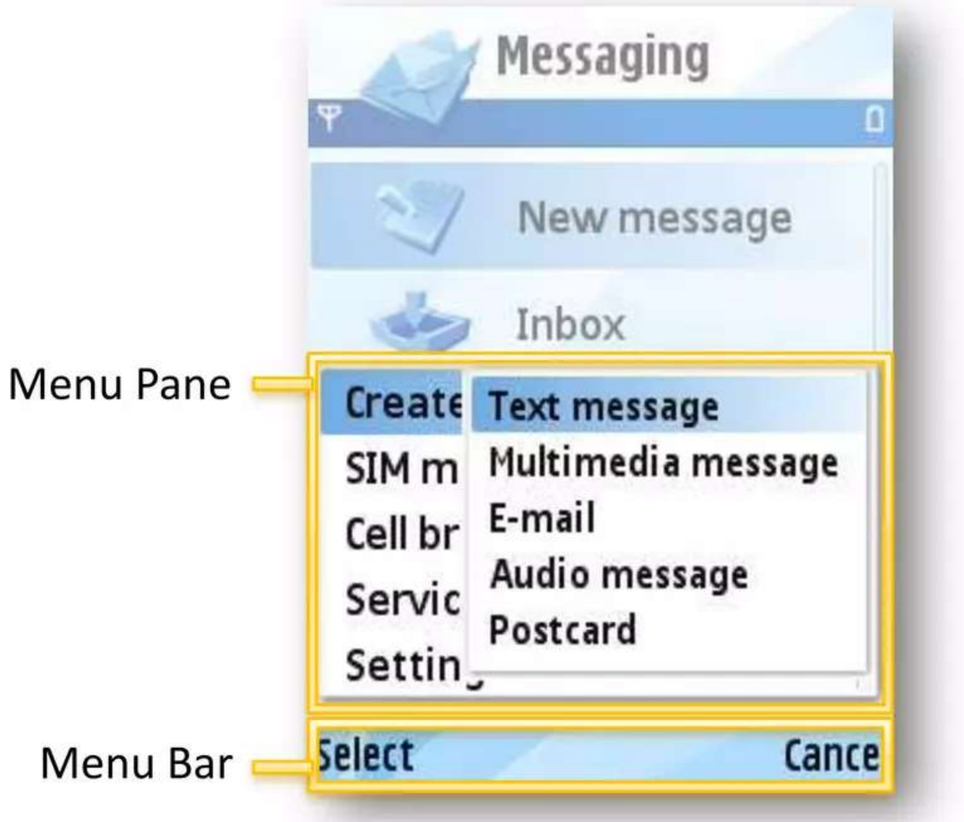
- **Symbian OS:** Locale Settings API

Defining and using

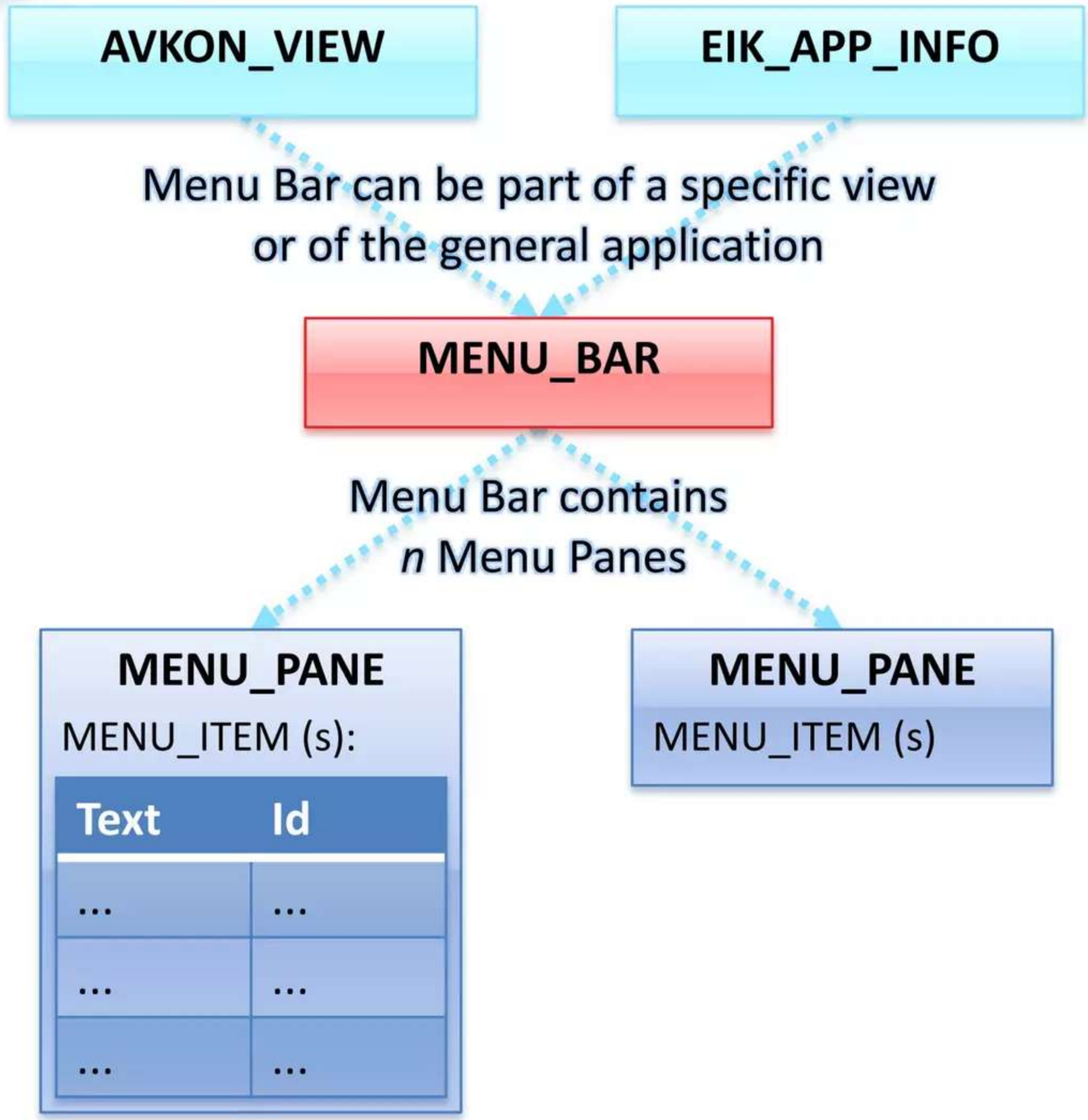
UI-Elements

Menus

- **Menu Bar**
  - Defines title of the menu (not shown in S60) and the contained menu panes (usually 1)
  - Can be default for Application or individual for each view
- **Menu Pane**
  - Defines unique section of a menu
  - Contains menu items, each with text string and command id



Hierarchical Structure



Hands-On: Menu

We won't use the UI designer in this example, but the wizard creates a view-based architecture

1. **Create a new project:**
   - **Name:** "UiTest"
   - **Template:** "S60 GUI Application with UI Designer"
   - **Design:** Empty

WARNING: Do not use the UI-Designer in this example, as it would override some of our own source code!

We want to do this in the first step...

Defining your menu

- Define the menu text in the .l01-file

UiTestContainer.l01

```
#define STR_ShowYourName "Show your name"
#define STR_Exit "Exit"
```

- Define the menu command ids in the .hrh-file

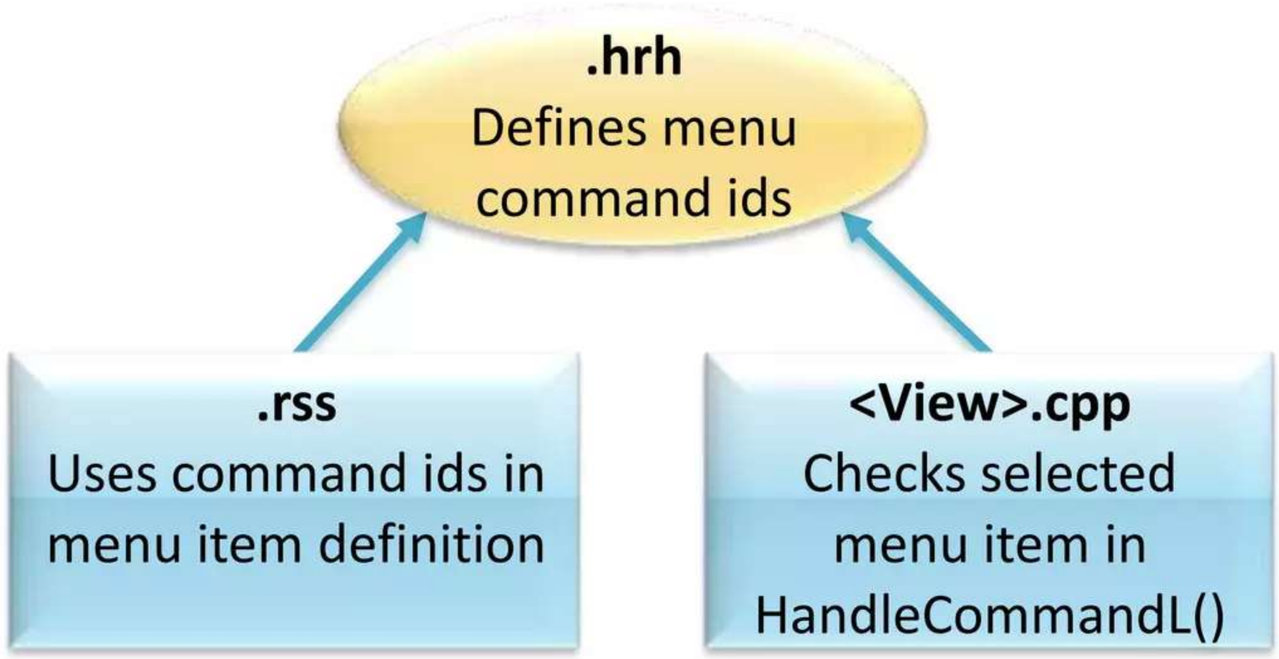
UiTestContainer.hrh

```
enum TUiTestContainerViewCommands
        {
        EUiTestCmdShowYourName = 0x6000,        // Any number that isn't 0
        EUiTestCmdExit                          // … enum automatically counts upwards
        };
```

Create this file in the inc-folder of your project

Menu Definition?

- **Menu command in the .hrh-file:**
  - Used in menu definition (.rss)
  - Used for checking which menu item was selected (<*View*>.cpp)



Defining your menu

- Create the menu in the .rss(i) file

UiTestContainer.rssi

```
#include "UiTestContainer.hrh"    // Add this at the beginning so that the command id is known
RESOURCE AVKON_VIEW r_ui_test_container_ui_test_container_view {
    cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT;
    menubar = r_ui_test_container_options_menu;
}
RESOURCE MENU_BAR r_ui_test_container_options_menu {
    titles = {
        MENU_TITLE { txt = "Example Menu"; menu_pane = r_ui_test_container_menu_pane; }
    };
}
RESOURCE MENU_PANE r_ui_test_container_menu_pane {
    items = {
        MENU_ITEM { command = EUiTestCmdShowYourName; txt = STR_ShowYourName; },
        MENU_ITEM { command = EUiTestCmdExit; txt = STR_Exit; }
    };
}
[...]
```

Menu Definition?

- **AVKON_VIEW** (S60 View-Architecture)
  - CBA: Defines Softkey labels
  - In this case Symbian OS standard-values (Options / Exit)
- **MENU_BAR**
  - Defines contained menu panes and their titles
- **MENU_PANE**
  - Defines individual items
  - Uses command ids from .hrh-file and text from .loc

Handling Commands

- View-based architecture, events are sent to:
  1. HandleCommandL() of the current view
  2. If unhandled, HandleCommandL() of the AppUi

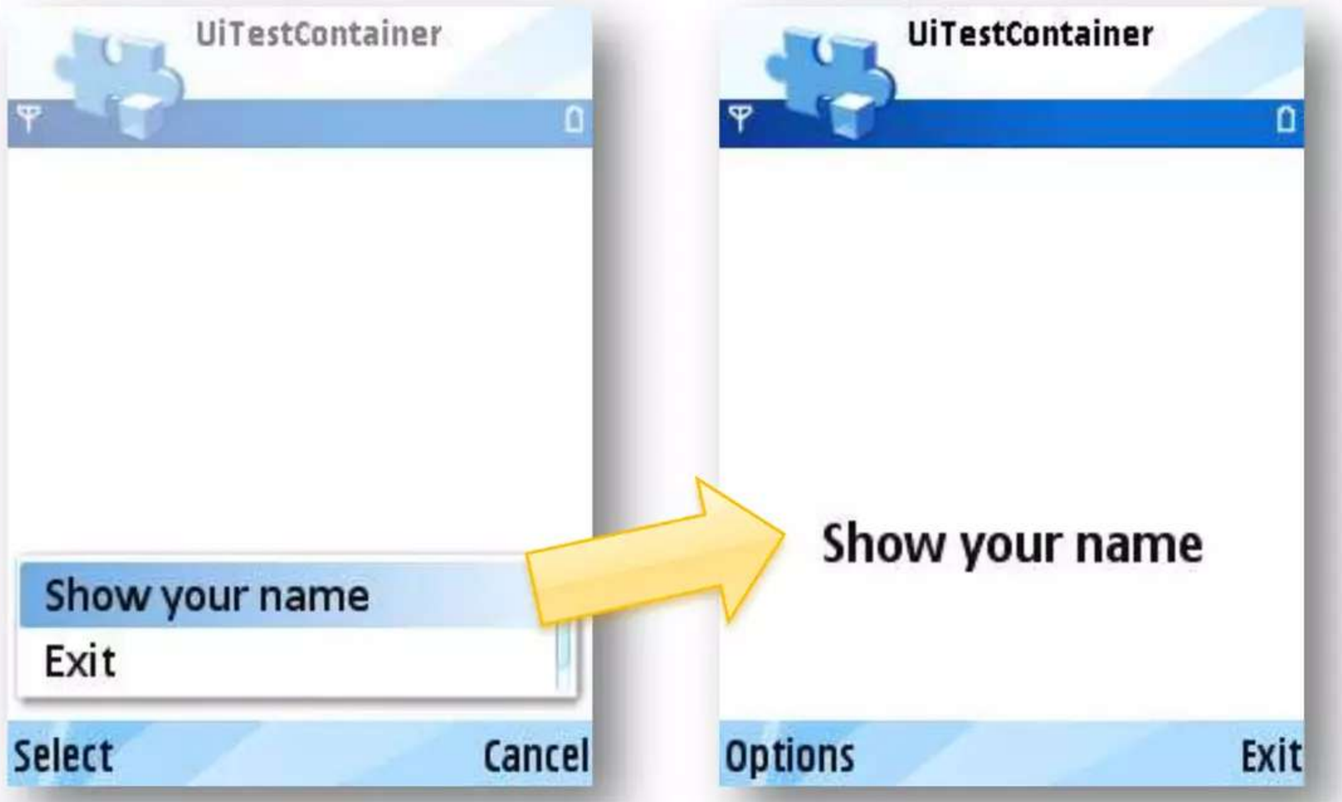
UiTestContainerView.cpp

```
#include "UiTestContainer.hrh"
void CUiTestContainerView::HandleCommandL( TInt aCommand ) {
   TBool commandHandled = EFalse;
   switch ( aCommand ) {
   case EUiTestCmdExit:
       // Exit the application when our new menu item is selected
      AppUi()->HandleCommandL( EEikCmdExit );
      break;
   default:
      break;
   }
   […]
}
```

Pre-defined system command. App. has to quit in any case, without a user query (e.g. forced shutdown)

Hands-On: Loading Strings

- **Task:** Load a String from a resource file and display it in a Label-control



Strings?

- **Hard-coding strings in source code:**

```
// Defining literals with the _LIT-macro
_LIT(msg, “Loading…”);
// The deprecated _L-macro
iEikonEnv->AlertWin(_L(“Starting…”));
```

- **… that’s bad! Why?**
  - Difficult to find and change later on
  - No clear application structure (mix source code with text)
  - Not localizable

Adding a Label-Control

- Label-Control declared in <*eiklabel.h*>
  (see SDK-help for CEikLabel – it's a Symbian OS control!)
- Define as private member of UiTestContainer.h:
  CEikLabel* iLabelName;
- Initialization code:

UiTestContainer.cpp

```
void CUiTestContainer::ConstructL (…) {
    […]
    iLabelName = new (ELeave) CEikLabel();
    iLabelName->SetContainerWindowL( *this );          // Label is non-window-owning
    iLabelName->SetTextL(KNullDesC);                   // Initially no text
    iLabelName->SetAlignment( EHCenterVCenter );       // Align the text in the center
    iLabelName->SetExtentToWholeScreen();              // Set the size and position
    […]
}
CUiTestContainer::~CUiTestContainer() {
        delete iLabelName;                             // Don't forget to delete the label!
}
```

Control-Management

- The Container has to manage the controls
  (→ see "*GUI Architectures*" module)
- Wizard generates "old", manual control-management
- Add the new label to the **enum**-control-counter

UiTestContainer.h

```
enum TControls {
    ELabelName,
    ELastControl
};
```

- Adapt the ComponentControl()-function

UiTestContainer.cpp

```
CCoeControl* CUiTestContainer::ComponentControl( TInt aIndex ) const {
switch ( aIndex ) {
case ELabelName:
    return iLabelName;
    break;
}
```

Modifying the Label-Text

- Create a function to modify the label text (define in .h)

UiTestContainer.cpp

```
void CUiTestContainer::SetLabelTextL( const TDesC& aText ) {
    iLabelName->SetTextL(aText);          // Update the text of the label
    DrawDeferred();                       // Request a screen redraw
}
```

- Define a text as a string resource

UiTestContainer.rssi

```
RESOURCE TBUF r_labeltext { buf = STR_ShowYourName; }
```

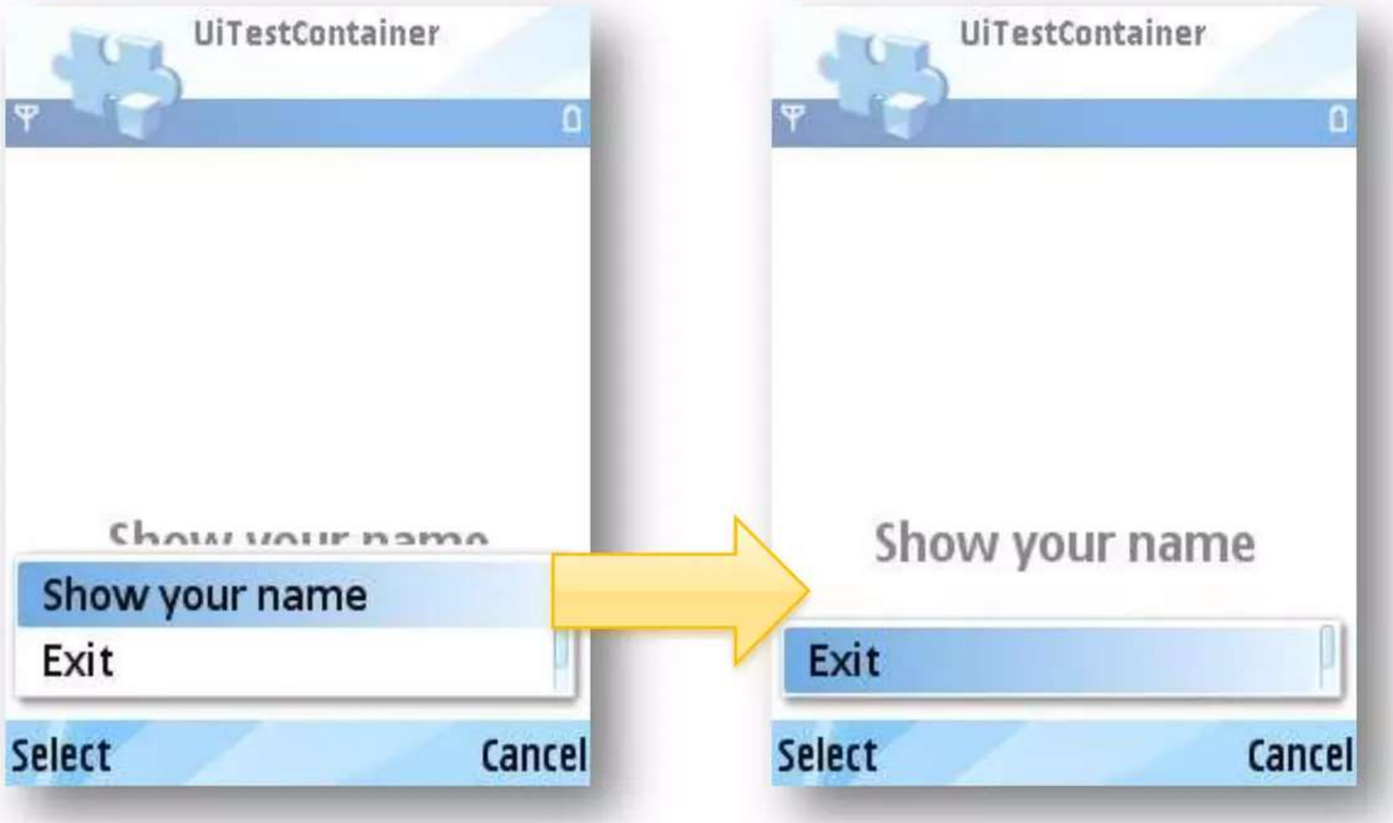
... this allows you to load the text (defined in .l01) into a descriptor from within the source code

Executing the Text-Change

- In CUiTestContainerView::HandleCommandL() for the other menu item:
- Use the **StringLoader** to load the text into a HBufC and call the iUiTestContainer->**SetLabelTextL()**-function (see Slide 17)
  - **Dereference** the HBufC with a * when using it as a TDesC& parameter

Hands-on: Dynamic Menu

- **Task:** Change menu content depending on app. state



Dynamic Menu

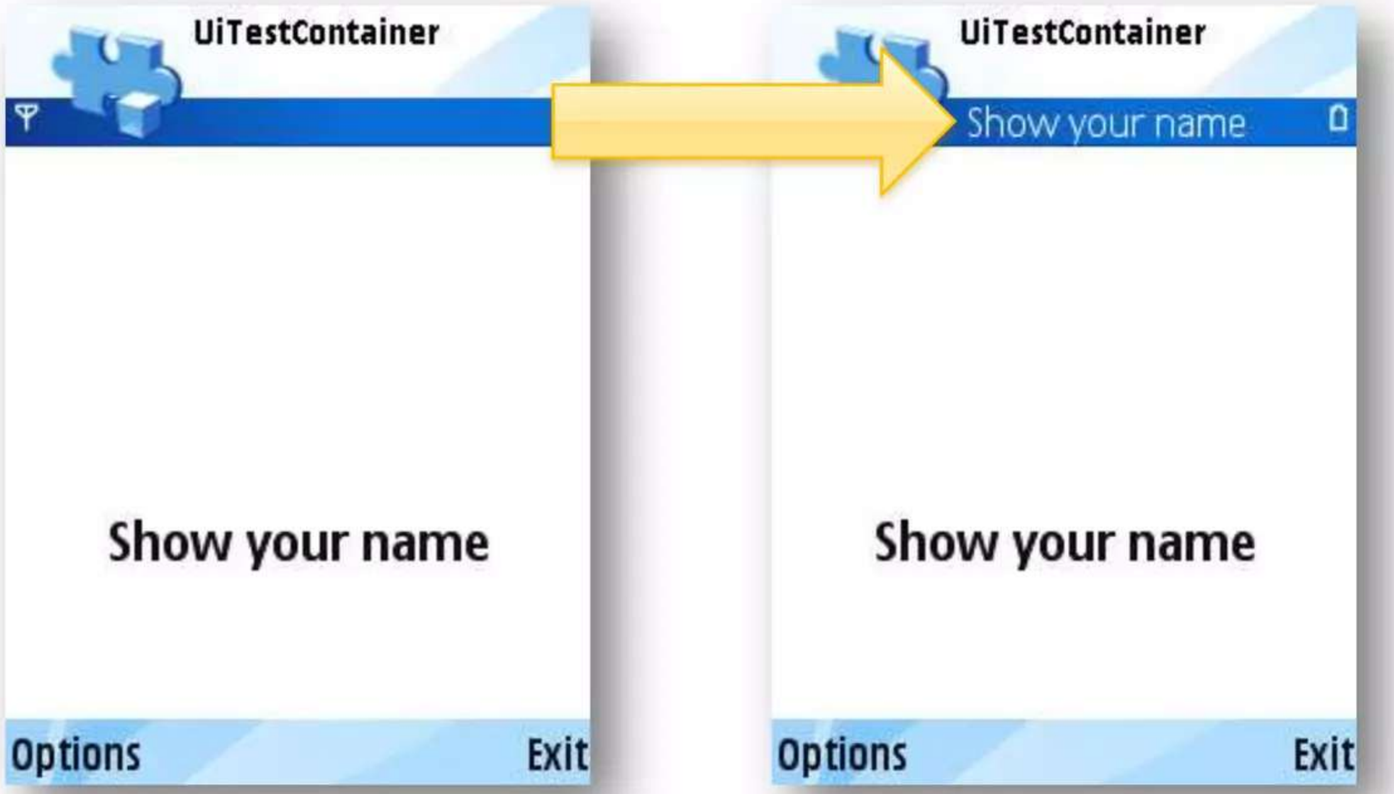
- **New TBool instance variable: iLabelTextSet**
  - Define it in UiTestContainerView (Default: EFalse)
  - Set to ETrue when menu item is selected
- **Override menu init-function from view base class**
  - Called after menu is read from resource file, but before it is displayed

UiTestContainerView.cpp

```
void CUiTestContainerView::DynInitMenuPaneL(TInt aResourceId, CEikMenuPane* aMenuPane) {
        // Check if the menu pane that is initialized is the one we are waiting for
        if (aResourceId == R_UI_TEST_CONTAINER_MENU_PANE) {
                // Dim the item depending on the application state
                aMenuPane->SetItemDimmed(EUiTestCmdShowYourName, iLabelTextSet);
        }
}
```

Hands-on: Navigation Pane

- Dynamically modify the S60 Navigation Pane



Accessing the Pane

1. **Access the status pane:**
   - **... in the AppUi:**
     CEikStatusPane* sp = StatusPane();
   - **... in an Avkon View:**
     CEikStatusPane* sp = iAvkonAppUi->StatusPane();
   - **... in most other places:**
     CEikStatusPane* sp = iEikonEnv->AppUiFactory()
     ->StatusPane();
   - **... for the rest:**
     CEikStatusPane* sp = CEikonEnv::Static()
     ->AppUiFactory()->StatusPane();
2. Access the required sub-pane (Title, Navigation, ...)

Navigation Decorator

- Navigation Decorator is a control (CCoeControl)
- **Options:** label, tab (group), image or indicator
- Owned by your own source code
  - **Private instance variable for CUiTestContainerView:**
    CAknNavigationDecorator* iNaviDecorator;
  - **Include required header:**
    #include <aknnavide.h> // Don't use *aknnavi.h*!
  - **Cleanup in Destructor:**
    delete iNaviDecorator;

Changing the Text

- Add this code to your own command handling (at some place where the descriptor still exists):

UiTestContainerView.cpp

```
// Access the status pane of the AppUi
CEikStatusPane* statusPane = iAvkonAppUi->StatusPane();
// Get the sub-pane, in our case the navigation pane
CAknNavigationControlContainer* naviPane = static_cast<CAknNavigationControlContainer*>
                    ( statusPane->ControlL(TUid::Uid(EEikStatusPaneUidNavi)) );

if (iNaviDecorator) {
    // Delete the old navi decorator first – if we have already created one
    delete iNaviDecorator;
    iNaviDecorator = NULL;          // Don't forget this - see the memory management course!
}
// Create a new navigation label control, based on our buffer
iNaviDecorator = naviPane->CreateNavigationLabelL(*buf);
// Push the new decorator onto the navigation pane stack
naviPane->PushL( *iNaviDecorator );
```

Various types of

Dialogs

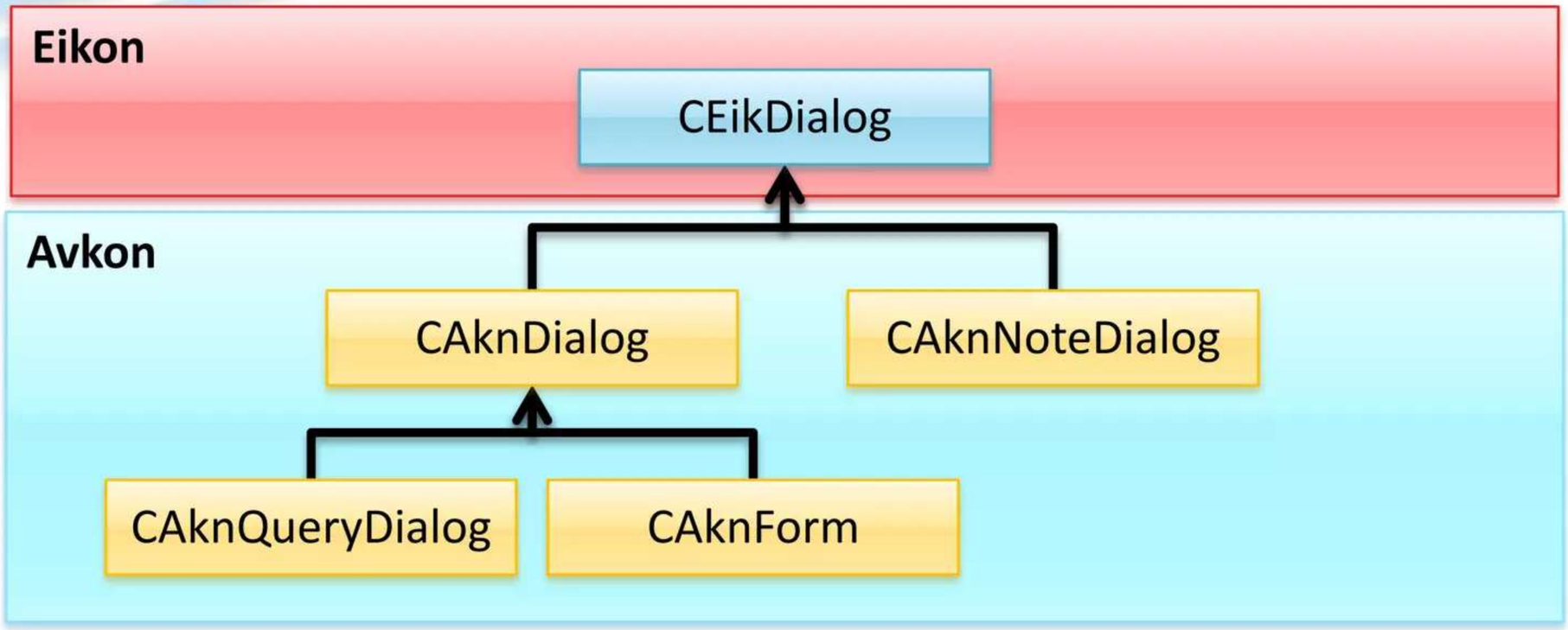
Dialogs?

- GUI application programming = managing controls and dialogs
- **Dialog:**
  - Pop-up window
  - Title
  - 1+ buttons (dismiss the dialog)
  - 1+ lines containing controls to display information
- Most dialogs are **modal**: foreground, get all user input until dismissed

Dialogs – Technical Overview

- **Dialog = Control**
  - Displays information
  - Allows user input
- **Contains 1+ Controls**
  - Each line has a control
  - Dialog class manages focus and sending key events!
- Derive from CAknDialog (S60: adds menu functionality) or CEikDialog

Dialog Classes (1)



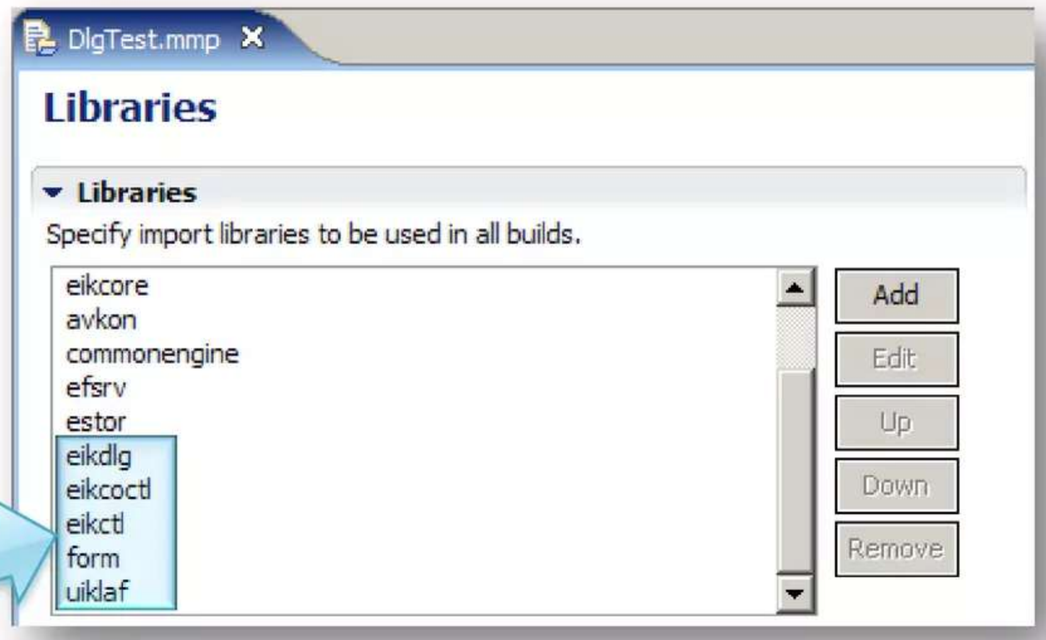
- **CEikDialog**
  - Base class for all dialogs
  - Can be derived from for implementing custom dialogs

Example: Custom Dialog

- Manual creation of a dialog
- Helps to understand process behind the scenes
  1. Define the **dialog contents** in the resource file
  2. Create a **class** that handles the user input for the dialog
  3. **Show** the dialog to the user

General Requirements

- Additional Libraries:



These libraries are required by this example

... how to find that out? Take a look at the SDK documentation of the classes we'll use: CEikDialog, CEikEdwin

1. DIALOG resource

- Resource file defines general appearance and contents of each line of the dialog

UiTest.rss

```
RESOURCE DIALOG r_dialog_test_dialog                // Define the layout of the control
{
    buttons = R_AVKON_SOFTKEYS_OK_CANCEL;           // Use S60 predefined softkeys for Ok and Cancel
    flags = EGeneralQueryFlags;                     // Define the general behaviour and layout of the dialog (-> avkon.hrh)
    items =
    {
        DLG_LINE                                    // The first dialog line: Text label
        {
            type=EEikCtLabel;                       // Add a control of the type: label (Symbian OS generic)
            id=ESimpleDlgTextLabel;                 // Each control has to have an ID (defined in .hrh)
            control=LABEL { txt="Enter text"; };    // Set parameters specific to the control specified in the “type”
        },                                          // Note: It’d be better to define the text in the .rls/.l01-file instead!
        DLG_LINE                                    // Second dialog line: Text editor
        {
            type=EEikCtEdwin;                       // Standard Symbian OS Editor-control
            id=ESimpleDlgText;
            control=EDWIN { width=10; maxlength=256; };  // Editor-specific settings
        }
    };
}
```

Resource Details: Dialog

- DIALOG is a STRUCT pre-defined in eikon.rh

eikon.rh

```
STRUCT DIALOG
{
    LONG flags=0;           // Characteristics of the dialog, e.g. EEikDialogFlagNoTitleBar
    LTEXT title="";         // Caption of the title bar (currently not used in S60)
    LLINK pages=0;          // For multipage-dialogs
    LLINK buttons=0;        // LLINK: Points to another resource. Use a predefined button/softkey-set or define your own (DLG_BUTTONS)
    STRUCT items[];         // Array of DLG_LINE-structures, specify contents of each dialog line
    LLINK form=0;
}
```

UiTest.rss

```
RESOURCE DIALOG r_dialog_test_dialog
{
    buttons = R_AVKON_SOFTKEYS_OK_CANCEL;          // Use S60 predefined softkeys for Ok and Cancel
    flags = EGeneralQueryFlags;
    items =
    {
        [...]
    };
}
```

On S60 this gets expaned to:
EEikDialogFlagWait |
EEikDialogFlagNoDrag |
EEikDialogFlagNoTitleBar |
EEikDialogFlagCbaButtons

Resource Details: Items

- DLG_LINEs specify contained control and its options:

UiTest.rss

```
DLG_LINE                                    // The first dialog line: Text label
{
    type=EEikCtLabel;                       // Add a control of the type: label (Symbian OS generic)
    id=ESimpleDlgTextLabel;                 // Each control has to have an ID (defined in .hrh)
    control=LABEL { txt="Enter text"; };    // Set parameters specific to the control specified in the “type”
},                                          // Note: It’d be better to define the text in the .rls/.l01-file instead!
DLG_LINE                                    // Second dialog line: Text editor
{
    type=EEikCtEdwin;                       // Standard Symbian OS Editor-control
    id=ESimpleDlgText;
    control=EDWIN { width=10; maxlength=100; }; // Editor-specific settings
}
```

eikon.rh

```
STRUCT EDWIN
{
    LONG flags=0;
    WORD width=0;
    WORD lines=1;
    WORD maxlength=0;
    // #define that adds S60 specific extensions:
    AKN_EDITOR_EXTENSIONS
}
```

UiTest.hrh

```
enum TUiTestIds
{
    […]                     // Command IDs, …
    ESimpleDlgTextLabel,    // Control ID for the label
    ESimpleDlgText          // Control ID for the editor window
};
```

2. Dialog Class

- Create an own class derived from CEikDialog
  - Initializes controls
  - Processes / saves control values when dismissed

SimpleDlg.h

```
class CSimpleDlg : public CEikDialog {
public:
    CSimpleDlg(TDes& aText);          // Parameter: Pre-filled text for the editor control
    ~CSimpleDlg();

private:                              // Inherited from CEikDialog
    void PreLayoutDynInitL();         // Initialize dialog's controls before the dialog is sized and laid out
    TBool OkToExitL(TInt aKeycode);   // You can check entered content and decide if the user may leave the dialog

private:
    TDes& iText;                      // Save a reference to the text of the editor control
};
```

Common Dialog Methods

- **PreLayoutDynInit()**
  - Called by the framework before the dialog is sized and layed out
  - Can be used to modify the layout or control contents
- **OkToExit()**
  - Called by the framework when user dismisses a dialog using OK
  - Extract data from dialog controls before it is destroyed
  - Check user input for validity
  - Dialog only closes if it returns ETrue

Class Initialization

- Constructor **saves reference** to the text buffer (pre-allocated by the caller)
- **Control()-method** of CEikDialog returns control of specified dialog line (→ DLG_LINE)
- Casted to **CEikEdwin** , in this case to **set the text**

SimpleDlg.cpp

```
CSimpleDlg::CSimpleDlg(TDes& aText) : iText(aText) {}

void CSimpleDlg::PreLayoutDynInitL() {
    // Set the text of the editor window to something pre-defined
    STATIC_CAST(CEikEdwin*, Control(ESimpleDlgText))->SetTextL(&iText);
}
```

Class Cleanup

- **OkToExit()** – only called when OK is selected
- Gives app. the chance to **check values**
  - **Get data** from the dialog and **save** it somewhere
  - **Return ETrue** if it's ok to dismiss the dialog, EFalse otherwise

SimpleDlg.cpp

```
CSimpleDlg::~CSimpleDlg() {}

TBool CSimpleDlg::OkToExitL(TInt aKeycode)
{
    // Get the current text of the editor-control and save it to the text buffer
    STATIC_CAST(CEikEdwin*,Control(ESimpleDlgText))->GetText(iText);
    return ETrue;          // Could do some more checks here and return EFalse
}
```

3. Launch the Dialog

- Important methods from the CEikDialog base class:
  - **PrepareLC(TInt aDialogResourceId)**
    - **Constructs the dialog** from the specified resource
  - **RunLD()**
    - **Runs the dialog**
    - Returns how the user **dismissed** the dialog (button ID)
    - **Destroys the dialog** on exit
    - **Returns** when dialog exits for waiting dialogs or immediately for non-waiting dialogs (e.g. progress notes)
  - **ExecuteLD()**
    - **Combination** of PrepareLC() and RunLD()

Execution

- e.g. in the menu item handler function of the current View / AppUi

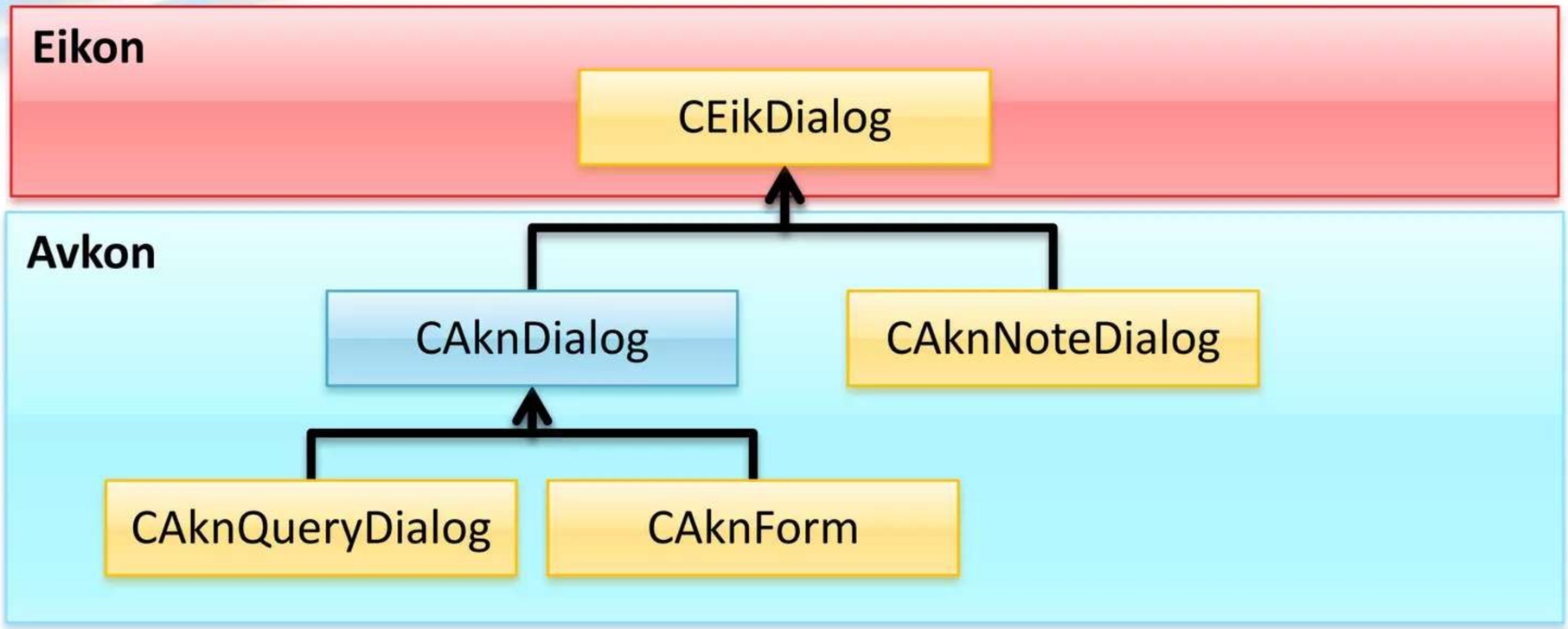
DlgTestAppUi.cpp

```
CSimpleDlg* dialog = new (ELeave) CSimpleDlg(iTmpTxt);
dialog->ExecuteLD(R_DIALOG_TEST_DIALOG);
```

Define iTmpText as instance variable in the AppUi, e.g. TBuf<100>

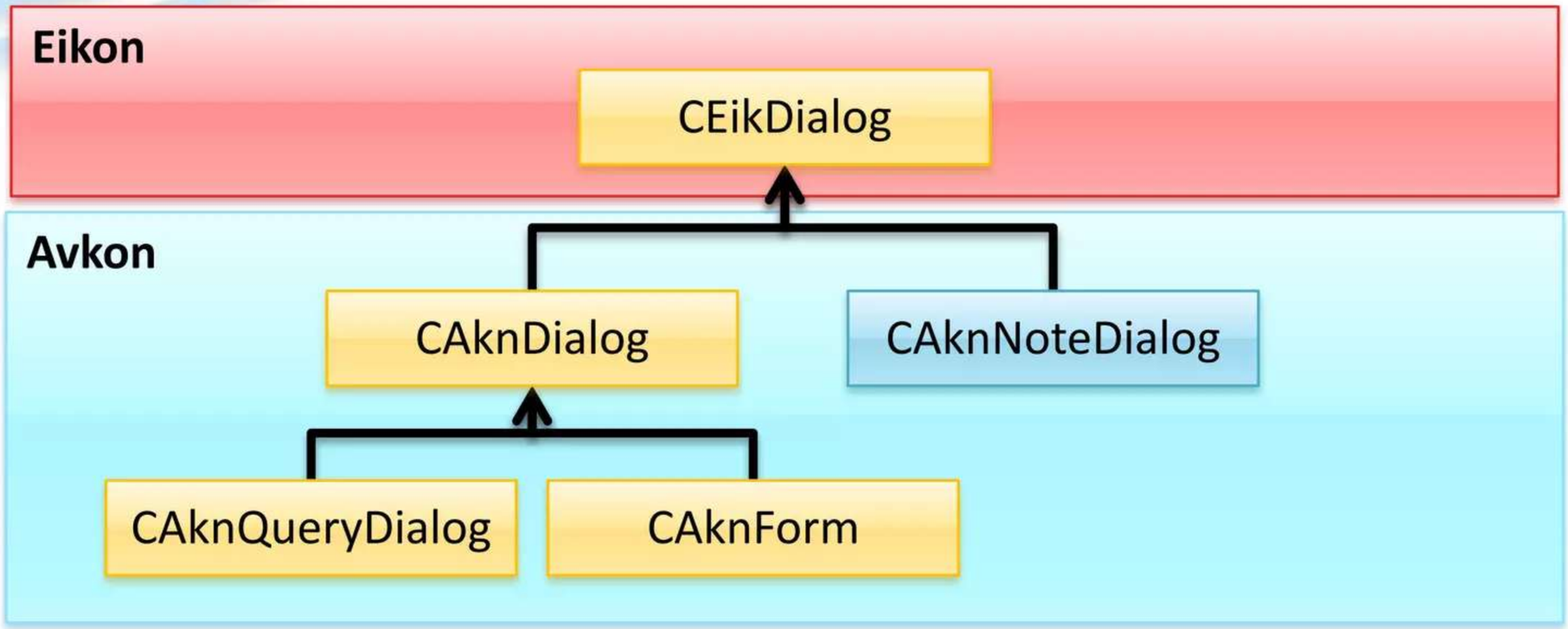
- Constructs the dialog, passes descriptor
- **ExecuteLD()**
  - **Returns ETrue if OK** is selected, EFalse for Cancel
  - **D** in the class name hints that function will **destroy this class after execution**
  - Pre-defined dialogs often put construction + execution into a single **RunDlgLD()-method**

Dialog Classes (2)



- **CAknDialog**
  - S60-specific
  - Adds menu functionality to the standard dialog

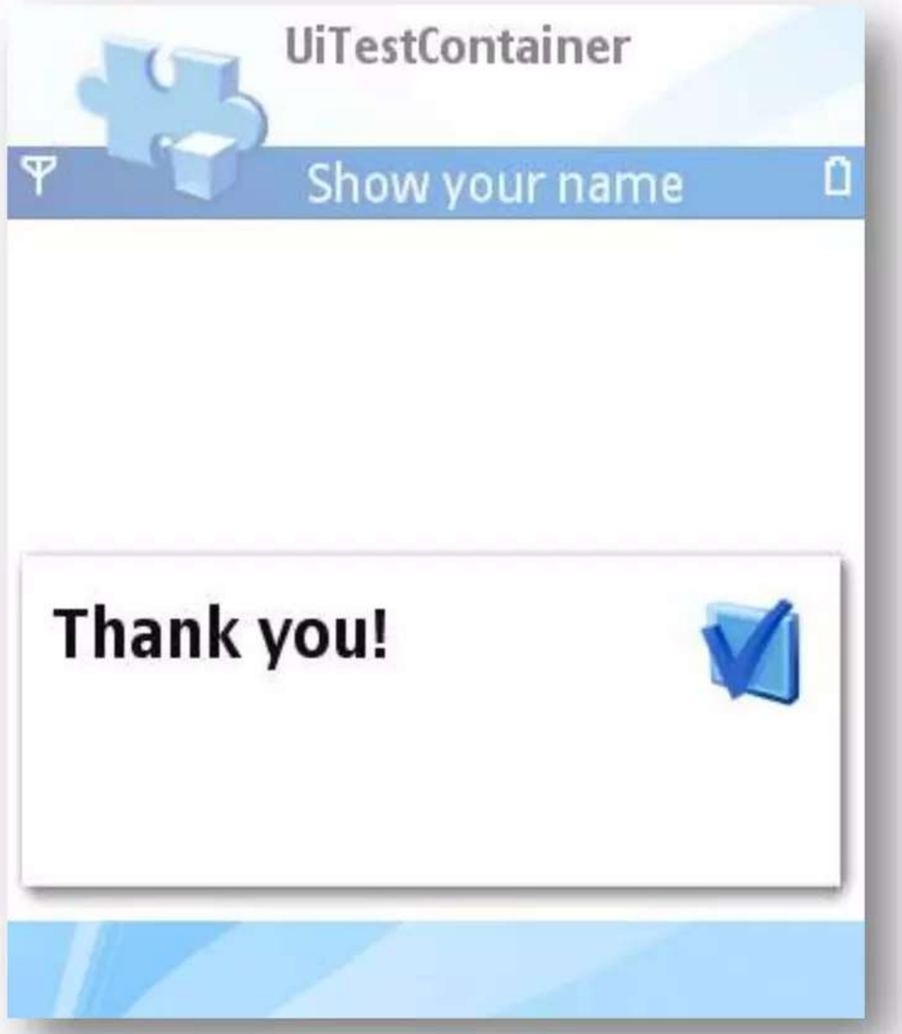
Dialog Classes (3)



- **CAknNoteDialog**
  - Displays information to the user
  - Further derivation for:
    - Wrapper classes (easier to use)
    - Progress dialogs

Hands-on: Note

- **Task:** Display a confirmation note after the user selected the menu entry



Notes

- **Inform the user about the current situation**

  - Text + graphical icon
  - Displayed for set time, dismiss themselves
  - Optional: sound
- **To create a note, either:**
  - **Use a Dialog derived from CAknNoteDialog**
    - Requires a DIALOG-resource in the .rss-file
  - **Use an AVKON (S60) defined wrapper class**
    - Resource predefined by AVKON → easier to use!

Notes

- Pre-defined notes; different durations, sounds, ...:
  - CAkn**ConfirmationNote**
    - Successfully completed process (e.g. settings saved)
  - CAkn**InformationNote**
    - Inform the user about an unexpected situation (e.g. non-serious error)
  - CAkn**WarningNote**
    - Warns the user of a situation that may require action (e.g. battery low)
  - CAkn**ErrorNote**
    - Warns the user of a situation that prevents processing to continue (e.g. wrong password)
      Use sparingly, prefer information notes
- **Other note types:** Progress Note, Wait Note (require resource definition)

Confirmation Note

- Show a confirmation note using the wrapper class:

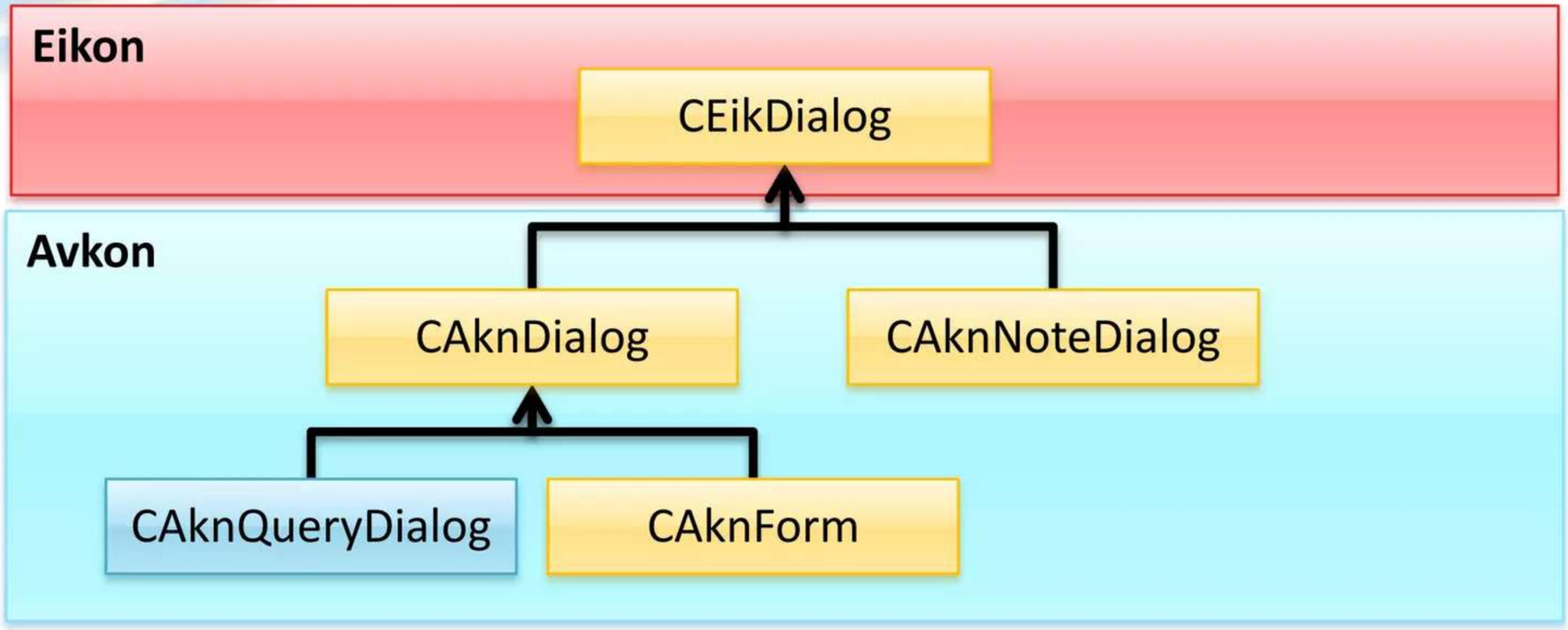
UiTestContainerView.cpp

```
#include <aknnotewrappers.h>

// Create a new confirmation note (pre-defined in S60 framework)
CAknConfirmationNote* noteConfirm = new (ELeave) CAknConfirmationNote;
// Displays and deletes the dialog – you don't have to delete it yourself!
noteConfirm->ExecuteLD(_L("Thank you!"));        // _L() for simplicity only – use dynamic text instead
```

- Similar code for other note types!
  - Just use different wrapper classes

Dialog Classes (4)



- **CAknQueryDialog**
  - Handles user input
  - Layout similar to notes, have to be dismissed by the user
  - Further derivation for specialized input
    - List-selection, confirmation, text, time, numbers, IP addresses, ...

Hands-on: Data Query

- Allow the user to enter his name:

Data Queries

- Enable various ways of **user input**
- Consist of a **prompt and input editor(s)**
- Several specialized data dialog classes defined in S60, e.g.:
  - **CAknTextQueryDialog:** Text input
  - **CAknNumberQueryDialog:** Allows entering a number
  - **CAknTimeQueryDialog:** Time/Date

Defining the Data Query

UiTestContainer.hrh

```
enum TUiTestContainerViewControls {
    ENameQueryDialogId = 1                    // Define a unique id for each control in the dialog
};                                            // ... important if you want to get a pointer to it in the source code
```

UiTestContainer.rssi

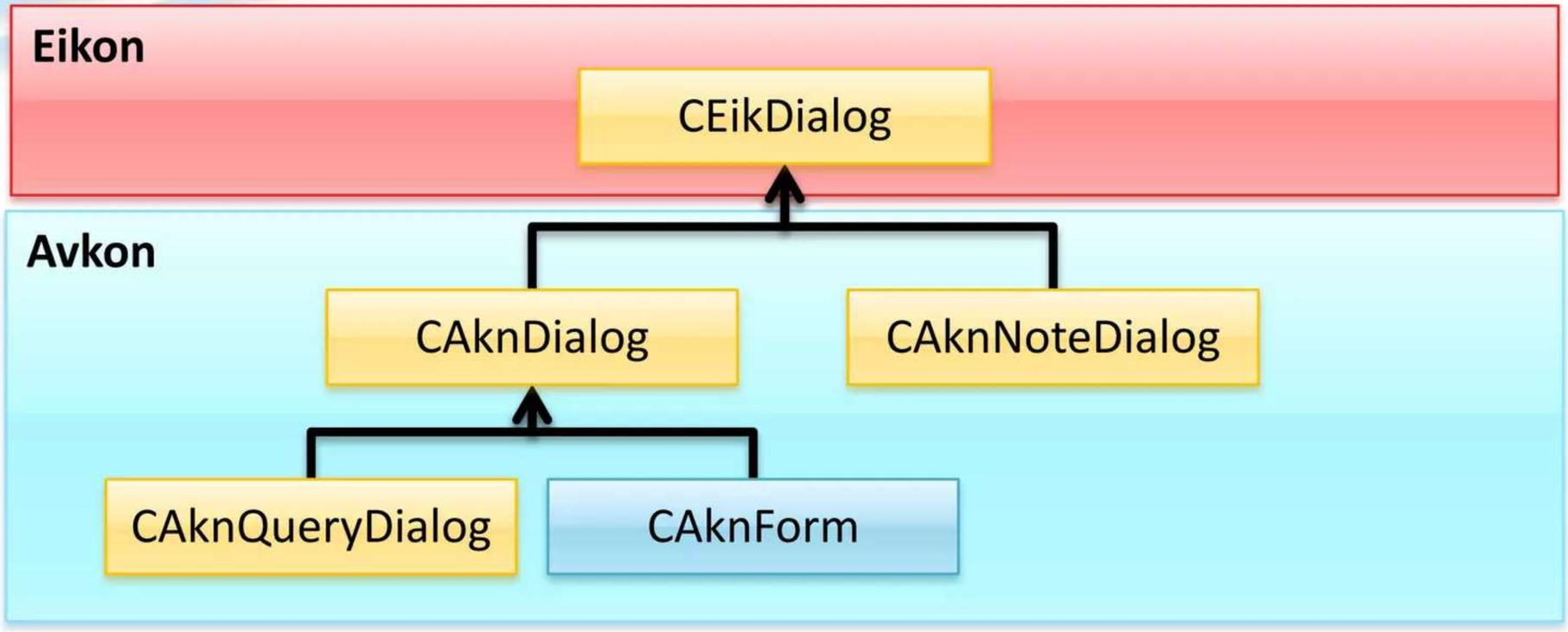
```
RESOURCE DIALOG r_ui_test_name_query
{
    flags = EAknGeneralQueryFlags;
    buttons = R_AVKON_SOFTKEYS_OK_CANCEL;          // Softkeys to be displayed
    items =
    {
        DLG_LINE                                   // Defines each line of a dialog – in this case the line is more complex
        {                                          // because the contained Avkon-control is made up of multiple lines itself
            type = EAknCtQuery;                    // Type of the query
            id = ENameQueryDialogId;               // Unique ID, defined by yourself in .hrh-file
            control = AVKON_DATA_QUERY
            {
                layout = EDataLayout;
                label = STR_NameQuery;             // Text of the query, is put on a predefined label-control -> define in .l01!
                control = EDWIN                    // Define the editor window control
                {
                   maxlength = 20;                 // Maximum allowed input length
                   default_input_mode = EAknEditorTextInputMode;        // Initial text input mode
                };
            };
        }
    };
}
```

Display the Data Query

UiTestContainerView.cpp

```
// Define the descriptor that is to contain the user name
TBuf<20> userName;
// Create the text query dialog, giving it a reference to the descriptor that will contain the data
CAknTextQueryDialog* txtDlg = CAknTextQueryDialog::NewL(userName);
// You can modify the dialog from the source code. This enables the T9 input method.
txtDlg->SetPredictiveTextInputPermitted(ETrue);
// Execute and destroy the dialog – you don't have to delete it yourself!
if (txtDlg->ExecuteLD(R_UI_TEST_NAME_QUERY))
{
    // Copy the old contents of our command handling code in here, so that it's only executed when the
    // user enters a name and doesn't choose to cancel.
}
```

Dialog Classes (5)



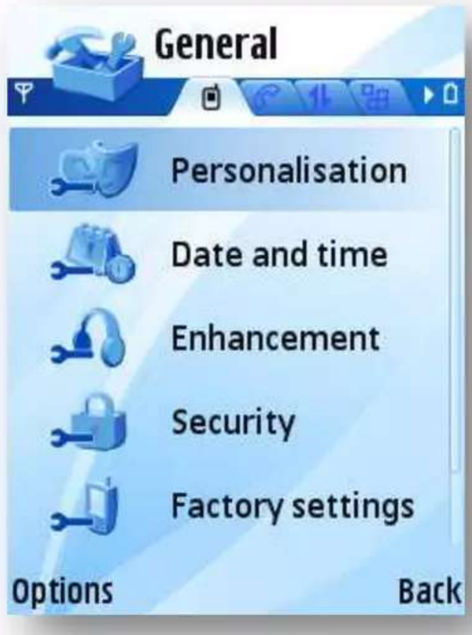
- **CAknForm**
  - Dual mode: viewing and editing
  - For collection of related data
  - Special form of a dialog

Forms

- **Two modes:**
  - **View mode** (= list box, highlight fields)
  - **Edit mode** (edit fields)
- Default: **Switch modes in options menu**
- **Multi-page Forms**: Tabs to group relevant controls

View State

Edit State



Multi-page Forms

Try it for your own

… let’s move to the Challenges!