

---

# Part 4: Recipes

---

<a href="#">Introduction</a> .....	1
<a href="#">2.1 Adding a Form</a> .....	3
<a href="#">2.2 Adding a button to a Form</a> .....	8
<a href="#">2.3 Adding softkeys to a Form</a> .....	15
<a href="#">2.4 Adding an option menu to a Form</a> .....	19
<a href="#">2.5 Display MessageBox to confirm an action</a> .....	25
<a href="#">2.6 Using a keypad control</a> .....	29
<a href="#">2.7 Get touch events</a> .....	39
<a href="#">2.8 Using base applications</a> .....	47
<a href="#">2.9 Get magnetic field from Compass Sensor</a> .....	56
<a href="#">2.10 Get geographic data from provider and show map</a> .....	62

---

# Introduction

**Note:** in this draft, pre-publication, and downloadable version of the book, we present the first 10 recipes only, as a taster of what's to come.

The recipes as well as this book and further resources can be accessed on our website <http://developer.bada.com>.

In Part 2 of the book we'll present 70 C++ recipes, organized into 9 functional groupings, that explore the bada APIs and provide you with ready-made code solutions for common \*how-to-do-it?\* questions. These are questions you are likely (in fact, certain!) to meet in the course of your own projects. By providing you with ready-made solutions, we want to help speed you on your way to your finished apps. Just as importantly, by solving these basic problems for you, we hope we will clear the way for you to focus on the interesting problems that your apps present - the kinds of features that differentiate your great idea from everyone else's.

## What's in the recipes

### 1. UI and extended UI

Basic recipes to get you started, demonstrating how to work with common UI elements including Frames, Forms, Controls, and Popups.

These recipes also include application input types, including keyboards, keys, touch, multi-touch, and gestures.

These recipes also include managing sensor and other hardware input including from the camera.

### 2. Fundamentals

Application basics, recipes that demonstrate how to work with the app lifecycle and with app state, and that demonstrate the use of bada idioms for C++, as well as common basic types from the bada class library, including Strings, Dates and Times, and Numbers.

These recipes also include using base (built-in) applications from your own apps.

### **3. Multimedia content**

Essential multimedia recipes.

### **4. Networking**

Recipes that demonstrate common networking, phone, messaging, and similar techniques.

### **5. Service-centric features**

Recipes that demonstrate interaction with server-based content.

### **6. Maps and locations**

Recipes that demonstrate how to use maps and location in your own apps.

### **7. Social networking and commerce**

Recipes that demonstrate how to use social networking services, including Facebook and Twitter, and other server-based content from within your own apps.

### **8. Security**

Recipes demonstrating security features and use.

### **9. Internationalization**

Recipes demonstrating internationalization features and use.

---

## 2.1 Adding a Form

### Problem description

You want to add a Form to a Frame-based application.

### The recipe

The following configuration is required for this recipe:

**Namespaces used:**

```
Osp::Ui, Osp::Ui::Controls
```

**Header files to #include:**

```
FUi.h, FUiControls.h
```

**Required libraries:**

```
FUi, FUiControls [.lib | .so]
```

**Required privilege level:**

```
Normal
```

**Required privilege groups:**

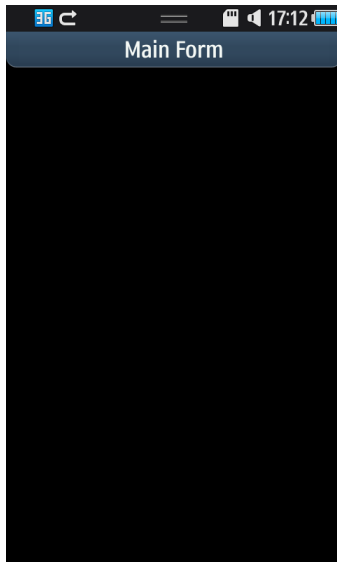
```
None
```

The top level window of an Application is called a Frame. A Frame is the ultimate parent object of all controls that an application possesses. A Frame can contain one or many Forms.

A Form is the highest level container. It is a full screen container which can contain user-created controls and system UI components to help provide functionality for an application.

The Frame and Form classes are provided by the `Osp::Ui::Controls` namespace.

This recipe shows two different methods that can be used to create a Form and add it to the Application's Frame. Each method details the steps required to add a Form to a Frame and provides some sample code snippets.



**Figure 2.1.1 A blank Form**

**Method 1: Adding a Form to the Frame using the `Osp::Ui::Controls::Form` class**

This simple recipe creates a new Form object using the system's `Osp::Ui::Controls::Form` class and adds it to the Application's Frame.

The Form should be created and added to the Application's Frame during the application initialization phase (`OnAppInitializing()`).

1. Create an instance of a Form and initialize it with the style parameters required to display a normal Form containing a title bar and an indicator bar.
2. Set the name of the Form and the text to be displayed in the Form's Title area.
3. Add the Form to the Application Frame.
4. Set the Form as the current form of the Frame.
5. Draw and Show the Form.

The following code snippet is an example of code that would be added to the Application's `OnAppInitializing()` method:

```
// Create a Form
Form* pForm = new Form();
pForm->Construct
    (FORM_STYLE_NORMAL|FORM_STYLE_TITLE|FORM_STYLE_INDICATOR);
pForm->SetName(L"MainForm");
pForm->SetTitleText(L"MainForm");

// Get a frame pointer and add the Form to the Frame
Frame* pFrame = GetAppFrame()->GetFrame();
pFrame->AddControl(*pForm);
// Set the current Form
pFrame->SetCurrentForm(*pForm);
// Call Draw & Show to display Form
pForm->Draw();
pForm->Show();
```

## Method 2: Adding form to the Frame using a user-defined Form class

This recipe creates the same results as in Method 1, but does it using a newly created Form class. This method may be more suitable for applications which contain multiple forms, as the creation of the Form's controls and its logic is implemented by the Form rather than in the Application code.

In this recipe the MainForm class is inherited from the base class `Osp::Ui::Controls::Form`.

The following code snippet shows an example of the MainForm class definition:

```
class MainForm :
    public Osp::Ui::Controls::Form
{
// Construction
public:
    MainForm(void);
    ~MainForm(void);
    bool Initialize(void);
// Call-back functions
public:
    result OnInitializing(void);
    result OnTerminating(void);
};
```

The Form should be created and added to the Application's Frame during the application initialization phase (`OnAppInitializing()`).

1. Create an instance of the `MainForm` type and initialise it.
2. Add the Form to the Application Frame.
3. Set the Form as the current form of the Frame.
4. Draw and Show the Form.

The following code snippet is an example of code that would be added to the Application's `OnAppInitializing()` method:

```
// Create a form
MainForm* pForm = new MainForm();
pForm->Initialize();

// Add the form to the frame
Frame* pFrame = GetAppFrame()->GetFrame();
pFrame->AddControl(*pForm);

// Set the current form
pFrame->SetCurrentForm(*pForm);

// Draw and Show the form
pForm->Draw();
pForm->Show();
```

The Form's style parameters and title text should be defined during initialization of the `MainForm` (`Initialize()`):

1. Call the base class constructor with the style parameters required to display a normal Form containing a title bar and an indicator bar.
2. Set the text to be displayed in the Form's Title area.

The following code snippet is an example of code that would be implemented in the `MainForm`'s `Initialize()` method:

```
Form::Construct(FORM_STYLE_NORMAL|FORM_STYLE_TITLE|FORM_STYLE_INDICATOR);  
SetTitleText(L"Main Form");
```

## **Hints, pitfalls, and related topics**

All containers, including the Form container, must be created on the heap memory. However, once the Form has been attached to the Frame, the application framework is responsible for its deletion when the application terminates. If the developer deletes the controls, it will cause a problem.

---

---

## 2.2 Adding a button to a Form

### Problem description

You want to add a Button control to a Form.

### The recipe

The following configuration is required for this recipe:

**Namespaces used:**

Osp::Ui, Osp::Ui::Controls, Osp::Media<sup>1</sup>, Osp::Graphics<sup>25</sup>

**Header files to #include:**

FUi.h, FUiControls.h, FMedia.h, FGraphics.h<sup>25</sup>

**Required libraries:**

FUi, FUiControls, FMedia<sup>25</sup>, FGraphics<sup>25</sup> [.lib | .so]

**Required privilege level:**

Normal

**Required privilege groups:**

IMAGE<sup>25</sup>

A Button is a control provided by the `Osp::Ui::Controls` namespace. A Button can vary in size and can contain either text or an image.

In order to handle the Button control's events, a listener must be used. The `IActionEventListener` is the interface class that should be implemented by any class interested in receiving a button's action events.

This recipe details the steps to add a Button control to a Form and describes the code that needs to be implemented for the Form to receive and handle a Button press event. Examples are provided for a button containing text and a button containing an image.

---

<sup>1</sup> For Button control displaying an image

This recipe assumes the steps detailed in method 2 of the ‘Adding a Form’ recipe [2.1] have been followed to add a Form to the Application’s Frame.

As shown in the ‘Adding a Form’ recipe the `MainForm` class is inherited from the `Form` base class. In order for this form to handle events from a control such as a button, an event listener interface has to be implemented. So in this recipe the `MainForm` class must implement the `Osp::Ui::IActionEventListener` interface class.

Because the `IActionEventListener` is shared by many controls, each control that uses this listener must be assigned a unique action ID. The action ID is used to identify the relevant control when handling events in the `OnActionPerformed()` event handler.

The following code snippet highlights the changes made to the `MainForm` class in order to handle a button control’s events:

```
class MainForm :
    public Osp::Ui::Controls::Form,
    public Osp::Ui::IActionEventListener
{
// Construction
public:
    MainForm(void);
    ~MainForm(void);
    bool Initialize(void);

// Implementation
protected:
    static const int ID_BUTTON_OK = 101;

public:
    result OnInitializing(void);
    result OnTerminating(void);
    void OnActionPerformed(const Osp::Ui::Control& source, int actionId); };
```

In order to handle the Button control action events the `IActionEventListener` class’s pure virtual function `OnActionPerformed()` needs to be implemented in the `MainForm` class. In this event

handler the control's event is identified by its action ID, which is registered when the control was created, and the relevant processing can be implemented.

The following code snippet is an example of the MainForm's OnActionPerformed() method:

```
void  
MainForm::OnActionPerformed(const Osp::Ui::Control& source, int actionId)  
{  
    switch(actionId)  
    {  
        case ID_BUTTON_OK:  
            {  
                // Add button handling functionality here  
            }  
            break;  
        default:  
            break;  
    }  
}
```

### Example 1: Creating a button containing text

In this example a button control with a text label is created on the Form.

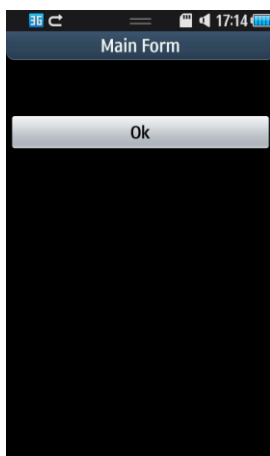


Figure 2.2.1 Button with text

The Button control should be added to the Form during initialization of the MainForm

(OnInitializing()):

1. Create a button object and initialize it by setting its size and position using a Rectangle instance.
2. Set the text string to be displayed on the button control.
3. Assign a unique action ID for the button control.
4. Register the IActionEventListener implemented by the MainForm class with the button control.
5. Add the button control to the form.

The following code snippet is an example of code that would be added to the MainForm's OnInitializing() method, in order to add a button control to the form:

```
// Create Button
Button* pButton = new Button();

// Set button size and position
pButton->Construct(Rectangle(10, 100, 460, 60));

// Set button text
pButton->SetText(L"Ok");

// Set button Action identifier
pButton->SetActionId(ID_BUTTON_OK);

// Register an event listener
pButton->AddActionEventListener(*this);

// Add the button control to the Main Form
AddControl(*pButton);
```

## Example 2: Creating a button containing an image

In this example a button control with an image is created on the Form.



Figure 2.2.2 Form with image

The Button control should be added to the Form during initialization of the `MainForm`

(`OnInitializing()`):

1. Create a `Button` object and initialize it by setting its size and position using a `Rectangle` instance.
2. Create an `Image` object and initialize.
3. Decode the image file into the decoded bitmap container, passing the local file path of the image file and colour format parameters.
4. Set the 'Normal' and 'Pressed' button images and position them at a `Point` relative to the top left corner of the `Button` `Rectangle`.
5. Assign a unique `action ID` for the button control.
6. Register the `IActionEventListener` implemented by the `MainForm` class with the button control.
7. Add the button control to the form.

The following code snippet is an example of code that would be added to the `MainForm`'s

`OnInitializing()` method, in order to add a button control to the form:

```
// Create Button
Button* pButton = new Button();

// Set button size and position
pButton->Construct(Rectangle(140, 100, 200, 200));

// Create a Bitmap and decode image
Bitmap* pBitmap = new Bitmap();
Image* pImage = new Image();
pImage->Construct();
pBitmap = pImage->DecodeN(L"/Res/bada.png",
    BITMAP_PIXEL_FORMAT_ARGB8888);

delete pImage;

// Set button images
pButton->SetNormalBitmap(Point(0, 0), *pBitmap);
pButton->SetPressedBitmap(Point(0, 0), *pBitmap);

delete pBitmap;

// Set button Action identifier
pButton->SetActionId(ID_BUTTON_OK);

// Register an event listener
pButton->AddActionEventListener(*this);

// Add the button control to the Main Form
AddControl(*pButton);
```

## Hints, pitfalls and related topics

All controls must be created on the heap memory. However, once a control has been added to the container, and the container attached to the Frame, the application framework is responsible for its deletion when the application terminates.

If a control is not added to a container, the programmer must explicitly delete it on termination of the application.

Controls, such as buttons, can be added and removed during run-time depending on UI context, to preserve memory. The methods `AddControl()` and `RemoveControl()` are provided to add and remove controls from a container.

## **Related recipes**

### 2.1 Adding a Form recipe

---

## 2.3 Adding softkeys to a Form

### Problem description

You need to add softkeys to a Form.

### The recipe

The following configuration is required for this recipe:

**Namespaces used:**

Osp::Ui, Osp::Ui::Controls, Osp::Media<sup>2</sup>, Osp::Graphics<sup>26</sup>

**Header files to #include:**

FUi.h, FUiControls.h, FMedia.h<sup>26</sup>, FGraphics.h<sup>26</sup>

**Required libraries:**

FUi, FUiControls, FMedia<sup>26</sup>, FGraphics<sup>26</sup> [.lib | .so]

**Required privilege level:**

Normal

**Required privilege groups:**

IMAGE<sup>26</sup>

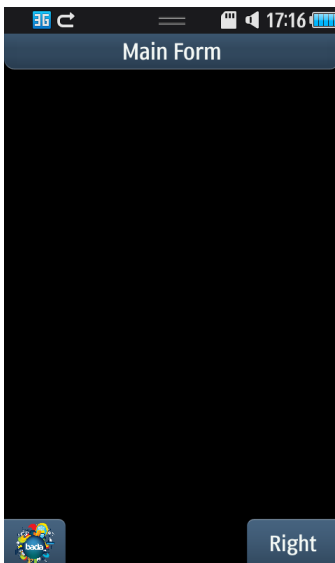
A Form is composed of an indicator area, tab and title areas, a client area, and a softkey area. Depending on the style of a form, it is determined whether an indicator, a tab, a title, an options key or the softkeys are shown.

A softkey can either display text or a bitmap. The softkey generates an action event when it is pressed. If an application wants to handle the action event generated by the softkey, it should implement an action event listener and register it to the softkey using `AddSoftkeyActionListener()` method.

---

<sup>2</sup> For softkey displaying an image

This recipe details the steps to create the left and right softkeys for a Form, and describes the code that needs to be implemented for the Form to receive and handle the softkey action events. In this recipe the left softkey contains an icon and the right softkey contains text.



**Figure 2.3.1 A Form with softkeys**

This recipe assumes the steps in method 2 of the ‘Adding a Form’ recipe (2.1) have been followed to add a Form to the Application’s Frame. It also assumes the steps described in the “Adding a button” recipe (2.2) have been followed to implement the `Osp::Ui::IActionEventListener` interface class.

To display the softkey area with a left and right softkey, the Form constructor needs to be called with the `FORM_STYLE_SOFTKEY_0` (related to left softkey) and `FORM_STYLE_SOFTKEY_1` (related to right softkey) style parameters set.

The following code snippet is an example of code that would be implemented in the MainForm’s `Initialize()` method:

```
Form::Construct(FORM_STYLE_NORMAL|FORM_STYLE_TITLE|  
FORM_STYLE_INDICATOR|FORM_STYLE_SOFTKEY_0|FORM_STYLE_SOFTKEY_1);
```

Action IDs need to be added to the MainForm for each of the softkeys, so that the softkey action events can be identified when handling events in the `OnActionPerformed()` event handler:

```
static const int ID_SOFTKEY_LEFT = 101;  
static const int ID_SOFTKEY_RIGHT = 102;
```

In the `OnActionPerformed()` event handler the softkey action events are identified by the action ID, which is registered when the softkey is created, and the relevant processing can be implemented.

The following code snippet is an example of the `MainForm`'s `OnActionPerformed()` method:

```
void  
MainForm::OnActionPerformed(const Osp::Ui::Control& source, int actionId)  
{  
    switch(actionId)  
    {  
        case ID_SOFTKEY_LEFT:  
            {  
                // Add left softkey handling code here  
            }  
            break;  
        case ID_SOFTKEY_RIGHT:  
            {  
                // Add right softkey handling code here  
            }  
            break;  
        default:  
            break;  
    }  
}
```

The softkeys should be created during initialization of the `MainForm` (`OnInitializing()`):

1. Create an `Image` object and initialize.
2. Decode the image file into the decoded bitmap container, passing the local file path of the image file and colour format parameters.
3. Set the left softkey to display an icon and assign the 'Normal' and 'Pressed' images.
4. Set the right softkey to display a text string.
5. Assign unique action IDs for both softkeys.
6. Register `IActionEventListener` implemented by the `MainForm` class with the softkeys.

The following code snippet is an example of code that would be added to the `MainForm`'s `OnInitializing()` method, in order to create the left and right softkeys:

```
// Create left softkey (SOFTKEY_0)
// Create a Bitmap and decode image
Bitmap* pBitmap = null;
Image* pImage = new Image();
pImage->Construct();
pBitmap = pImage->DecodeN(L"/Res/bada.png",
    BITMAP_PIXEL_FORMAT_ARGB8888);

SetSoftkeyIcon(SOFTKEY_0, *pBitmap, pBitmap);

delete pImage;
delete pBitmap;

// Create right softkey (SOFTKEY_1)
SetSoftkeyText(SOFTKEY_1, "Right");

// Assign action IDs for both the softkeys
SetSoftkeyActionId(SOFTKEY_0, ID_SOFTKEY_LEFT);
SetSoftkeyActionId(SOFTKEY_1, ID_SOFTKEY_RIGHT);

// Register event listener
AddSoftkeyActionListener(SOFTKEY_0, *this);
AddSoftkeyActionListener(SOFTKEY_1, *this);
```

## Hints, pitfalls and related topics

If both an icon and text are set for the same softkey at the same time, the text takes precedence over the icon. A softkey can be enabled or disabled using the `SetSoftkeyEnabled()` method, with the enable parameter set to either true or false.

## Related recipes

2.1 Adding a Form recipe

2.2 Adding a Button recipe

---

## 2.4 Adding an option menu to a Form

### Problem description

You want to add an options menu to a Form and get the user's selections.

### The recipe

The following configuration is required for this recipe:

**Namespaces used:**

Osp::Ui, Osp::Ui::Controls

**Header files to #include:**

FUi.h, FUiControls.h

**Required libraries:**

FUi, FUiControls [.lib | .so]

**Required privilege level:**

Normal

**Required privilege groups:**

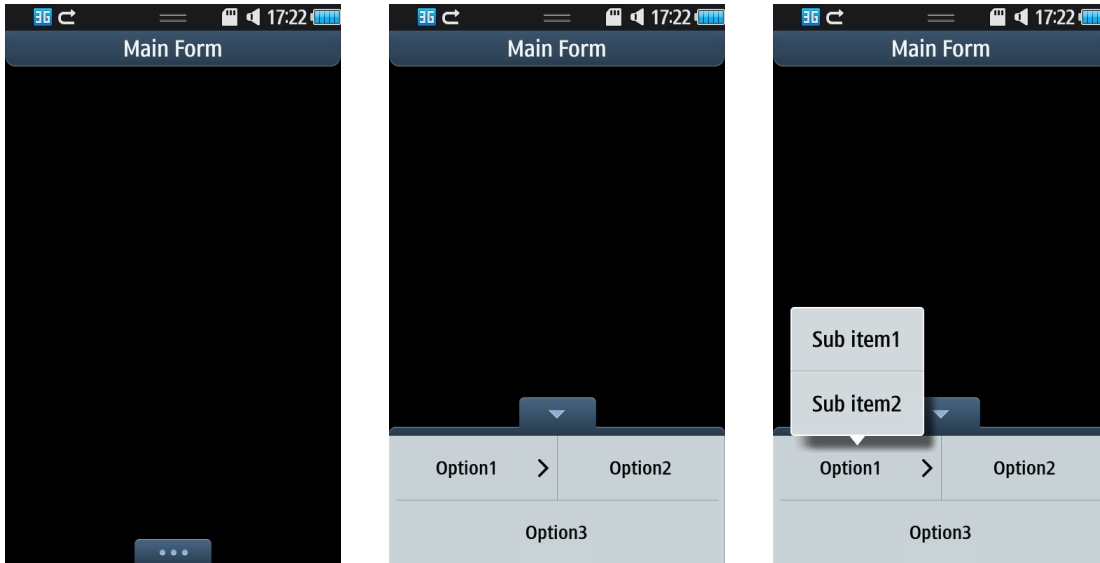
None

A Form is composed of an indicator area, tab and title areas, a client area, and a softkey area. Depending on the style of a form, it is determined whether an indicator, a tab, a title, an options key or the softkeys are shown.

An Option Menu is a menu control activated from an Option Key located in the command area of the display. It can be used to present multiple options to the user, and is displayed over the content of the form. It is a hierarchical menu, which can contain up to 2 levels, consisting of main items and sub items. The menu items can only be presented as text strings, bitmaps cannot be displayed.

An Option Menu control does not get added to a container, so it must be explicitly deleted when it is no longer required. The Application Framework will not delete the Option Menu when the application terminates.

This recipe details the steps to create the Option Key and add a 2-level Option Menu to a Form, and describes the code that needs to be implemented for the Form to receive and handle the Option Key and Option Menu action events.



**Figure 2.4.1 Option Menu with 2-levels**

This recipe assumes the steps detailed in method 2 of the ‘Adding a Form’ recipe (2.1) were followed to add a Form to the Application’s Frame. It also assumes the steps described in the ‘Adding a button’ recipe (2.2) have been followed to implement the `Osp::Ui::IActionEventListener` interface class.

To display the Option Key in the softkey area of the display, the Form constructor needs to be called with the `FORM_STYLE_OPTIONKEY` style parameter set.

The following code snippet is an example of code that would be implemented in the MainForm’s `Initialize()` method:

```
Form::Construct(FORM_STYLE_NORMAL|FORM_STYLE_TITLE|  
FORM_STYLE_INDICATOR|FORM_STYLE_OPTIONKEY);
```

An Action ID needs to be added to the MainForm for the Option Key, so that its action event can be identified when handling events in the OnActionPerformed() event handler:

```
static const int ID_OPTIONKEY = 100;
```

Similarly, each Option Menu main item and sub item requires an Action ID to identify it in the OnActionPerformed() event handler:

```
static const int ID_OPTION1 = 103;  
static const int ID_OPTION2 = 104;  
static const int ID_OPTION3 = 105;  
static const int ID_OPTIONA = 106;  
static const int ID_OPTIONB = 107;
```

In the OnActionPerformed() event handler the Option Key and Option Menu item action events are identified by the action ID, which are registered when the Option Key and Option Menu are created, and the relevant processing can be implemented.

When the event with the action ID associated with the Option Key is received, the Option Menu is activated and displayed using the SetShowState() and Show() methods.

The following code snippet is an example of the MainForm's OnActionPerformed() method:

```
void  
MainForm::OnActionPerformed(const Osp::Ui::Control& source, int actionId)  
{  
    switch(actionId)  
    {  
        case ID_OPTIONKEY:  
            {  
                // Display the OptionMenu  
                __pOptionMenu->SetShowState(true);  
                __pOptionMenu->Show();  
            }  
    }  
}
```

```
    }
    break;

case ID_OPTION1:
    // Because this item has sub items attached, you
    // don't need to do any additional event handling here
    break;

case ID_OPTION2:
    // Add event handling code here
    break;

case ID_OPTION3:
    // Add event handling code here
    break;

case ID_OPTIONA:
    // Add event handling code here
    break;

case ID_OPTIONB:
    // Add event handling code here
    break;

default:
    break;
}
}
```

The Option Key and Option Menu should be created during initialization of the MainForm (OnInitializing()):

1. Set the Action ID of the Form's Option Key control.
2. Register the `IActionEventListener` implemented by the MainForm class.
3. Create an Option Menu and initialize.
4. Add the main items to the Option Menu, specifying the text string to be displayed and the item's action IDs.
5. Add any sub items to the Option Menu, specifying the index of the main item to attach the sub item to, the text string to be displayed and the item's action IDs.
6. Register the `IActionEventListener` implemented by the MainForm class, with the Option Menu object.

The following code snippet is an example of code that would be added to the `MainForm`'s `OnInitializing()` method, in order to create the Option Key and Option Menu:

```
// Create an OptionKey
SetOptionkeyActionId(ID_OPTIONKEY);
AddOptionkeyActionListener(*this);

// Create an OptionMenu
__pOptionMenu = new OptionMenu();
__pOptionMenu->Construct();
__pOptionMenu->AddItem("Option1", ID_OPTION1);
__pOptionMenu->AddItem("Option2", ID_OPTION2);
__pOptionMenu->AddItem("Option3", ID_OPTION3);

// Add a sub items to the first item in the options menu @ index 0
__pOptionMenu->AddSubItem(0, "Sub item1", ID_OPTIONA);
__pOptionMenu->AddSubItem(0, "Sub item2", ID_OPTIONB);

// Register an event listener
__pOptionMenu->AddActionEventListener(*this);
```

## Hints, pitfalls and related topics

The example shown in this recipe has a 2-level fixed structure for the Option Menu's menu hierarchy. It is also possible to dynamically add, change and delete items during runtime. If you want to change the content of the Option Menu there are several methods that can be used to add, change and delete both main items and sub items. These methods are: `AddItem()`, `AddSubItem()`, `SetItemAt()`, `SetSubItemAt()`, `RemoveItemAt()` and `RemoveSubItemAt()`.

If you need to determine if there are any sub items associated with a main item, and how many there are, the `GetSubItemCount()` method can be used.

In the example, the first main item in the menu (`ID_OPTION1`) has sub items associated with it. In this scenario no code needs to be implemented in the `OnActionPerformed()` event handler for this `Action ID`, as the display of the sub items is automatically performed by the Option Menu object. But,

if there is code for this `Action ID` in the `OnActionPerformed()` event handler, it will get executed after the sub items are displayed.

When a sub item, or main item with no associated sub item is selected by the user, the Option Menu object will generate the appropriate event for the form to handle and the menu will be removed from the display.

Because of the way in which the Option Menu is removed from the display when it has no sub items to display, it is not recommended to dynamically create any sub items for a main item on selection of that main item.

If the items text string is too long, it will be truncated when displayed:

- A main item will get truncated at 2 lines of text (in English) and '...' will be added to the end of the string
- A sub item will get truncated at 1 line of text (in English) and '...' will be added to the end of the string.

## **Related recipes**

2.1 Adding a Form recipe

2.2 Adding a Button recipe

---

## 2.5 Display MessageBox to confirm an action

### Problem description

You want to prompt the user for confirmation of an action, outside the fixed view or menu structure of your application.

### The recipe

The following configuration is required.

**Namespaces used:**

```
Osp::Ui, Osp::Ui::Control
```

**Header files to #include:**

```
FUi.h, FUiControls.h
```

**Required libraries:**

```
FUi [.lib | .so], FUiControls [.lib | .so]
```

**Required privilege level:**

```
Normal
```

**Required privilege groups:**

```
None
```

This recipe shows how to launch and display an acknowledgement MessageBox that displays a message to the user and an OK button, and waits for the user to confirm by pressing OK. A MessageBox can also have no button, in which case the message is simply displayed, or two or three buttons. Buttons have fixed text, set by the `MSGBOX_STYLE` parameter.

Handling the user response is easy – it is set in the argument passed to the launch method.

Unlike other Control-derived classes which are used as building blocks to populate the Forms from which an application's UI is constructed, a MessageBox is a so-called *modal* control; a MessageBox is instantiated, initialized, and displayed outside the normal event flow of the currently displayed Form and any Controls the Form owns.

Unlike other controls, `MessageBox` derives from the `Window` class, via the derivation chain:

```
Object -> Control -> Container -> Window -> [ MessageBox | Popup ]
```

A `MessageBox` therefore is drawn directly into the window area it defines, *on top of* the current Form.

Conceptually therefore a `MessageBox` defines an additional mode of interaction outside the fixed infrastructure of the application's Forms; a `MessageBox` is direct and immediate.

To use a `MessageBox`, #include the following header files and declare the appropriate namespace:

```
#include <FUi.h>
#include <FUiControls.h>
#include <FBase.h>

using namespace Osp::Ui::Controls;
```

The code to instantiate and launch a `MessageBox` is almost trivial:

```
/*
 * Giving a title and message for the popup windows showing to the user
 */
void Utils::showMessage(const String &title, const String &message)
{
    MessageBox messageBox;
    messageBox.Construct(title, message, MSGBOX_STYLE_OK, 3000);
    // Call ShowAndWait - draw, show itself and process events
    int modalResult = 0;
    messageBox.ShowAndWait(modalResult);
    switch(modalResult)
    {
        case MSGBOX_RESULT_OK:
            // TODO
            break;
        default:
            break;
    }
}
```

The `MessageBox` constructor is used to set values for the title, message, style, and timeout period of the instance; and then the `ShowAndWait (result)` method is called, which instantiates and displays the `MessageBox`, sets the user response in the result argument, and manages destruction and cleanup.

## Hints, pitfalls, and related topics

Forms and menus create the fixed architecture of an interactive application. But a typical application also requires the more dynamic kinds of interaction provided by prompts, alerts, messages and notifications based on the specific context of the moment.

The `MessageBox` class is almost trivial to use, but is rather limited in the options it provides; in particular, only a small selection of preset buttons is provided:

<code>MSGBOX_STYLE_NONE</code>	Contains no buttons
<code>MSGBOX_STYLE_OK</code>	Contains one button: OK
<code>MSGBOX_STYLE_CANCEL</code>	Contains one button: CANCEL
<code>MSGBOX_STYLE_OKCANCEL</code>	Contains two buttons: OK and CANCEL
<code>MSGBOX_STYLE_YESNO</code>	Contains two buttons: YES and NO
<code>MSGBOX_STYLE_YESNOCANCEL</code>	Contains three buttons: YES, NO and CANCEL
<code>MSGBOX_STYLE_ABORTRETRYIGNORE</code>	Contains three buttons: ABORT, RETRY, and IGNORE
<code>MSGBOX_STYLE_CANCELTRYCONTINUE</code>	Contains three buttons: CANCEL, TRY and CONTINUE
<code>MSGBOX_STYLE_RETRYCANCEL</code>	Contains two buttons: RETRY and CANCEL

Where more flexibility is required, the `Popup` class enables a similar style of dynamic interaction, and is fully customizable – but also requires explicit construction, initialization, display, and cleanup. `Popups` can also contain buttons, labels, and other simple controls to enable capturing a richer variety of user choices or selections.

On return of the `ShowAndWait ()` method (which is a synchronous function), the result argument contains the result of the call.

The following results are returned by a `MessageBox`:

<code>MSGBOX_RESULT_CLOSE</code>	The message box was closed
<code>MSGBOX_RESULT_OK</code>	The OK button was selected
<code>MSGBOX_RESULT_CANCEL</code>	The cancel button was selected
<code>MSGBOX_RESULT_YES</code>	The Yes button was selected
<code>MSGBOX_RESULT_NO</code>	The No button was selected
<code>MSGBOX_RESULT_ABORT</code>	The Abort button was selected
<code>MSGBOX_RESULT_TRY</code>	The Try button was selected
<code>MSGBOX_RESULT_RETRY</code>	The Retry button was selected
<code>MSGBOX_RESULT_IGNORE</code>	The Ignore button was selected
<code>MSGBOX_RESULT_CONTINUE</code>	The Continue button was selected

The `MessageBox` class is easy to use, but it has one important limitation.

- The call to launch a `MessageBox` is synchronous and blocking; the `MessageBox` is not dismissed until the user response is received or until it times out.

For this reason a `MessageBox` should never be launched from a listener callback method, since it will block the callback's return and disrupt the asynchronous event flow. Therefore, inside a callback method, always use a `Popup` instead.

---

## 2.6 Using a keypad control

### Problem description

You want to pop up a Keypad and get input from it.

### The recipe

The following configuration is required for this recipe:

**Namespaces used:**

Osp::Ui, Osp::Ui::Controls, Osp::Base

**Header files to #include:**

FUi.h, FUiControls.h, FBase.h

**Required libraries:**

FUi, FUiControls, FBase [.lib | .so]

**Required privilege level:**

Normal

**Required privilege groups:**

None

A Keypad is a UI control which provides a means for the user to input alphanumeric text.

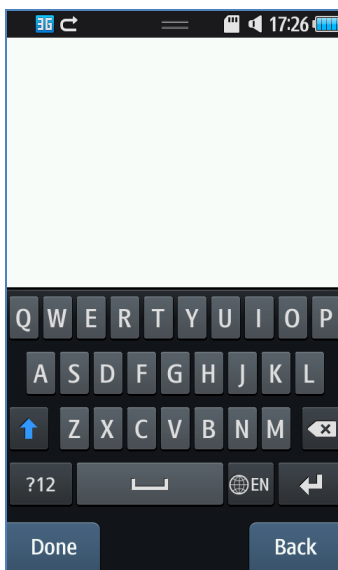
A Keypad can either be used within an application as a stand-alone keypad or as part of an Edit Field or Edit Area input control. For a stand-alone keypad, only a full screen keypad is available. For a keypad automatically created by the system for an Edit Field or Edit Area control, both full screen and overlay keypads are supported

This recipe shows two different methods that can be deployed to make use of the Keypad control. Each method details the steps required to create and use the Keypad, and provides some sample code snippets.

Both methods in this recipe assume the steps detailed in method 2 of the ‘Adding a Form’ recipe have been followed to add a Form to the Application’s Frame.

### Method 1: Using the stand-alone keypad control

This recipe creates a stand-alone keypad using the `Osp::Ui:Controls::Keypad` class and displays it in the Application Frame. Any text events generated by the keypad, can be processed by the application by implementing the `Osp::Ui::ITextEventListener` interface class.



**Figure 2.6.1 Stand-alone keypad**

When a stand-alone keypad is used, it is not added to any container, so it must be explicitly deleted when it is no longer required. The Application Framework will not delete the Keypad when the application terminates.

The `ITextEventListener` interface receives text events. In order for the application to retrieve any text that has been entered by the user using the Keypad control, this interface must be implemented.

The following code snippet shows an example of the `MainForm` class definition:

```
class MainForm :
    public Osp::Ui::Controls::Form,
    public Osp::Ui::ITextEventListener
{
// Construction
public:
    MainForm(void);
    ~MainForm(void);
    bool Initialize(void);

private:
    Osp::Ui::Controls::Keypad *__pKeypad;

public:
    result OnInitializing(void);
    result OnTerminating(void);

    void ShowKeypad(bool show);

    void OnTextValueChanged(const Osp::Ui::Control& source);
    void OnTextValueChangeCanceled(const Osp::Ui::Control& source);
};
```

The Keypad control can be created during initialization of the MainForm (OnInitializing()):

1. Create a keypad object.
2. Set the required style and input mode category parameters e.g. normal style with alphabetic input mode.
3. Register the ITextEventListener implemented by the MainForm class to the Keypad control.

The following code snippet is an example of code that would be added to the MainForm's OnInitializing() method, in order to create a Keypad control:

```
// Create the Keypad member object
__pKeypad = new Keypad();

// Initialize with specified style and category
__pKeypad->Construct(KEYPAD_STYLE_NORMAL, KEYPAD_MODE_ALPHA);
```

```
// Register an event listener
__pKeypad->AddTextEventListener(*this);
```

As the Keypad control is not explicitly added to the Form container, it will not automatically be displayed. The keypad's Show() method must be called to display the keypad. For example, the display of the keypad could be triggered by a press of a softkey invoking the ShowKeypad() method:

```
void
MainForm::ShowKeypad(bool show)
{
    if(true == show)
    {
        __pKeypad->Show();
    }
    else // call the Form's Show() method
    {
        Show();
    }
}
```

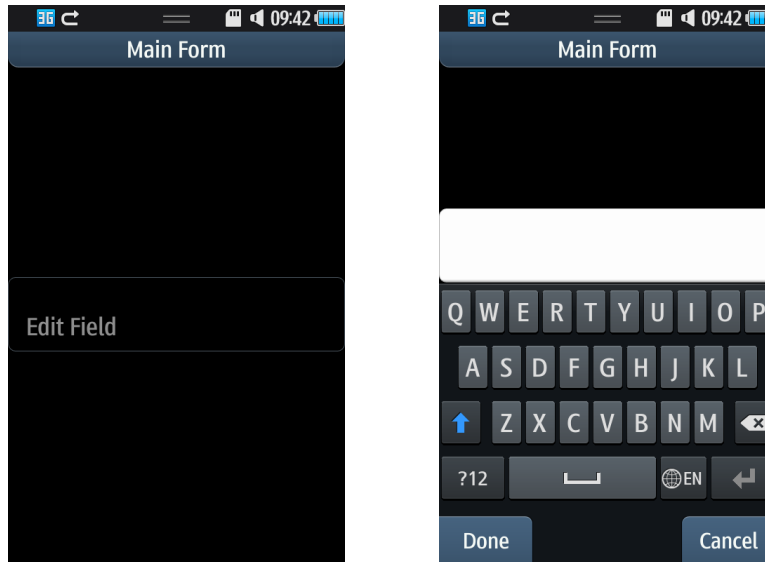
The stand-alone keypad displays 'Done' and 'Cancel' command buttons in the softkey area. Although they look like softkeys, they are not softkeys and do not behave in the same way as softkeys. When the 'Done' button is tapped, a ITextEventListener action event is triggered and the MainForm's OnTextValueChanged() method is invoked. When the 'Cancel' button is tapped, a ITextEventListener action event is triggered and the MainForm's OnTextValueChangedCanceled() method is invoked.

These methods can be used by an application to retrieve the data that has been entered by the user using the Keypad control. For example the OnTextValueChanged() method can be implemented to retrieve the text string from the keypad object:

```
void
MainForm::OnTextValueChanged(const Osp::Ui::Control& source)
{
    String string(__pKeypad->GetText()); // Retrieve text from Keypad object
}
```

## Method 2: Using an overlay style keypad within an edit field

This recipe creates an edit field, which displays one line of editable text to the user. It is implemented in the overlay style, so that when the user taps on the edit control, an overlay keypad is launched and the edit control is automatically scrolled so that it does not overlap with the overlay keypad, below.



**Figure 2.6.2 Edit Field with overlay keypad**

In order to use the overlay style of keypad, the Form must contain a scroll panel and the edit field control must be added to the scroll panel. This recipe creates a scroll panel container using the `Osp::Ui::Controls::ScrollPanel` class and creates an edit field control using the `Osp::Ui::Controls::EditField` class.

The `IScrollPanelEventListener` interface must be implemented in order to handle the events generated when the keypad is opened or closed, e.g. for retrieval of the edited text string. The `ITextEventListener` interface must be implemented if the application needs to be aware of when the cursor moves within the edit field. The `IActionEventListener` interface must be implemented if the edit field is created with the optional keypad command button controls, in order to handle the events generated when these buttons are tapped.

The following code snippet shows an example of the MainForm class definition:

```
class MainForm :
    public Osp::Ui::Controls::Form,
    public Osp::Ui::IActionEventListener,
    public Osp::Ui::ITextEventListener,
    public Osp::Ui::IScrollPanelEventListener
{
    // Construction
public:
    MainForm(void);
    ~MainForm(void);
    bool Initialize(void);

    // Implementation
protected:
    static const int ID_BUTTON_EDITFIELD_DONE = 100;
    static const int ID_BUTTON_EDITFIELD_CANCEL = 101;

private:
    Osp::Ui::Controls::EditField* __pEditField;
    Osp::Ui::Controls::ScrollPanel* __pScrollPanel;
    Osp::Base::String* __pEditFieldText;

public:
    result OnInitializing(void);
    result OnTerminating(void);

    void OnActionPerformed(const Osp::Ui::Control& source,
        int actionId);

    void OnTextValueChanged(const Osp::Ui::Control& source);
    void OnTextValueChangeCanceled(const Osp::Ui::Control& source);

    void OnOverlayControlCreated(const Osp::Ui::Control& source);
    void OnOverlayControlOpened(const Osp::Ui::Control& source);
    void OnOverlayControlClosed(const Osp::Ui::Control& source);
    void OnOtherControlSelected(const Osp::Ui::Control& source);
};
```

The `ScrollPanel` container and the `EditForm` control can be created during initialization of the `MainForm` (`OnInitializing()`):

1. Create `ScrollPanel` object and initialize it by setting its size and position using a `Rectangle` instance.
2. Create `EditField` object and initialize it with its size, position, edit field style, input style, title, input length limit and table view style parameters.
3. Register the `IActionEventListener` implemented by the `MainForm` class with the `EditField` control.
4. Register `ITextEventListener` implemented by the `MainForm` class with `EditField` control.
5. Register the `IScrollPanelEventListener` implemented by the `MainForm` class with the `EditField` control.
6. Add the `EditField`'s left and right command buttons.
7. Create a `String` object to store the edited text.
8. Add the `EditField` control to the `ScrollPanel` container.
9. Add the `ScrollPanel` container to the `MainForm`.

The following code snippet is an example of code that would be added to the `MainForm`'s `OnInitializing()` method, in order to create a overlay style edit field control:

```
// Create ScrollPanel
__pScrollPanel = new ScrollPanel();
__pScrollPanel->Construct(Rectangle(0, 0, 480, 712));

// Create EditField
__pEditField = new EditField();
__pEditField->Construct(Rectangle(0, 300, 480, 80),
    EDIT_FIELD_STYLE_NORMAL, INPUT_STYLE_OVERLAY,
    true, 20, GROUP_STYLE_SINGLE);
__pEditField->SetGuideText(L"Edit Field");

// Register event listeners
__pEditField->AddActionEventListener(*this);
__pEditField->AddTextEventListener(*this);
__pEditField->AddScrollPanelEventListener(*this);

// Add command buttons
__pEditField->SetOverlayKeypadCommandButton(
    COMMAND_BUTTON_POSITION_LEFT, L"Done",
    ID_BUTTON_EDITFIELD_DONE);
__pEditField->SetOverlayKeypadCommandButton(
```

```
COMMAND_BUTTON_POSITION_RIGHT, L"Cancel",
ID_BUTTON_EDITFIELD_CANCEL);

// Create String to store entered text
__pEditText = new String();

// Add an EditField to the Scroll Panel
__pScrollPanel->AddControl(*__pEditField);

// Add a ScrollPanel to the Form
AddControl(*__pScrollPanel);
```

When the user taps on the edit field control the overlay keypad is launched. In order to receive and process any changes to the overlay keypad, the following `IScrollPanelEventListener` methods should be implemented:

`OnOtherControlSelected()`, `OnOverlayControlClosed()`, `OnOverlayControlCreated()` and `OnOverlayControlOpened()`.

The following code snippet shows an example of how the edit field's text string can be stored in a string member variable when the keypad is opened, in the `OnOverlayControlOpened()` method:

```
void
MainForm::OnOverlayControlOpened(const Osp::Ui::Control& source)
{
    __pEditText->Clear();
    __pEditText->Append(__pEditField->GetText());
}
```

When the overlay keypad is active and the user taps either of the edit field control's command buttons, an action event is generated and is handled in the `OnActionPerformed()` method. The following code snippet is an example of the command button event handling in the `MainForm`'s

`OnActionPerformed()` method:

```
void
MainForm::OnActionPerformed(const Osp::Ui::Control& source, int actionId)
{
    switch(actionId)
    {
        case ID_BUTTON_EDITFIELD_DONE:
            __pScrollPanel->CloseOverlayWindow();
            __pEditField->Draw();
            __pEditField->Show();
            break;
        case ID_BUTTON_EDITFIELD_CANCEL:
            __pEditField->SetText(*__pEditFieldText);
            __pScrollPanel->CloseOverlayWindow();
            __pEditField->Draw();
            __pEditField->Show();
            break;
        default:
            break;
    }
}
```

When the ‘Done’ button is tapped the overlay keypad is closed and the Edit Field is redrawn. When the ‘Cancel’ button is tapped the original text string, which is stored in the member variable `__pEditFieldText`, is loaded into the edit field, the overlay keypad is closed and Edit Field redrawn.

## Hints, pitfalls and related topics

The `ITextEventListener` interface receives text events, but the type of text events it receives will differ depending on what type of UI control it is registered with. For example, in method 1 the `OnTextValueChanged()` method is invoked when the ‘Done’ button is tapped, but in method 2 it is invoked each time the cursor moves within the edit field. By default an overlay style keypad is not displayed with any command buttons. If these buttons are required they must be created using the `SetOverlayKeypadCommandButton()` method.

When initializing a `ScrollPanel` object for the `EditField` to be placed within, the location and size parameters for the `ScrollPanel` should be set relative to the top left of the `Form`’s available client

area. So if a `Form` contains an indicator area and/or a title area, the heights of these areas should be taken into account when setting the `ScrollPane`'s height parameter. If these values are not set correctly the overlay keypad will not be displayed in the correct position.

To clear any previous text entered in the stand-alone keypad, the `SetText ()` method can be called with an empty string parameter.

## **Related recipes**

### 2.1 Adding a Form recipe

---

## 2.7 Get touch events

### Problem description

You want to retrieve touch events from the user input.

### The recipe

The following configuration is required for this recipe:

**Namespaces used:**

Osp::Ui, Osp::Ui::Controls, Osp::Graphics<sup>3</sup>

**Header files to #include:**

FUi.h, FUiControls.h, FGraphics.h<sup>27</sup>

**Required libraries:**

FUi, FUiControls, FGraphics<sup>27</sup> [.lib | .so]

**Required privilege level:**

Normal

**Required privilege groups:**

None

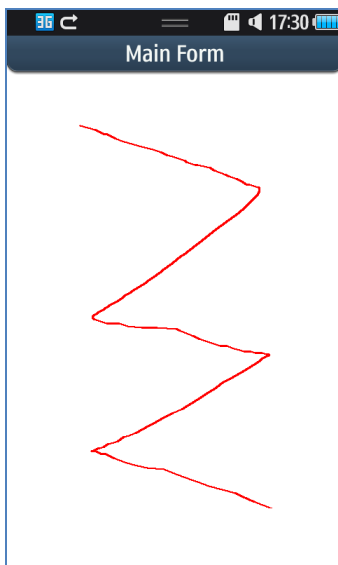
The bada platform supports both single-point and multi-point touch user input. This recipe details the steps to enable single-point touch user input within a Form and describes the code that needs to be implemented for the Form to receive and handle touch events.

This recipe assumes the steps detailed in method 2 of the ‘Adding a Form’ recipe have been followed to add a Form to the Application’s Frame.

---

<sup>3</sup> For graphics canvas drawing in the recipe

In this recipe the touch screen user input is handled by the application and is used to draw lines on a graphics canvas. A line is drawn on the graphics canvas following the motion of the users touch input. The double-tap touch input is used to clear the canvas.



**Figure 2.7.1 Drawing lines with touch screen input**

As shown in the “Adding a Form” recipe the MainForm class is inherited from the Form base class. In order for this form to handle events from the touch screen, an event listener interface has to be implemented. So in this recipe the MainForm class must implement the `Osp::Ui::ITouchEventListener` interface class.

The following code snippet shows an example of the MainForm class definition:

```
class MainForm :
    public Osp::Ui::Controls::Form,
    public Osp::Ui::ITouchEventListener
{
public:
```

```
MainForm(void);
~MainForm(void);
bool Initialize(void);

private:
    Osp::Graphics::Canvas* __pDrawingCanvas;
    Osp::Graphics::Point* __pStartPoint;

public:
    result OnDraw(void);
    result OnInitializing(void);
    result OnTerminating(void);

    void OnTouchPressed(const Osp::Ui::Control& source,
        const Osp::Graphics::Point& currentPosition,
        const Osp::Ui::TouchEventInfo & touchInfo);
    void OnTouchLongPressed(const Osp::Ui::Control& source,
        const Osp::Graphics::Point& currentPosition,
        const Osp::Ui::TouchEventInfo & touchInfo);
    void OnTouchReleased(const Osp::Ui::Control& source,
        const Osp::Graphics::Point& currentPosition,
        const Osp::Ui::TouchEventInfo & touchInfo);
    void OnTouchMoved(const Osp::Ui::Control& source,
        const Osp::Graphics::Point& currentPosition,
        const Osp::Ui::TouchEventInfo & touchInfo);
    void OnTouchDoublePressed(const Osp::Ui::Control& source,
        const Osp::Graphics::Point& currentPosition,
        const Osp::Ui::TouchEventInfo & touchInfo);
    void OnTouchFocusIn(const Osp::Ui::Control& source,
        const Osp::Graphics::Point& currentPosition,
        const Osp::Ui::TouchEventInfo & touchInfo);
    void OnTouchFocusOut(const Osp::Ui::Control& source,
        const Osp::Graphics::Point& currentPosition,
        const Osp::Ui::TouchEventInfo & touchInfo);
};
```

In order to receive touch events, the `MainForm` must register the touch event listener using the `AddTouchEventListener()` method.

The creation of the canvas and the registration of the touch event listener should be performed during initialization of the `MainForm` (`OnInitializing()`):

1. Create a graphics canvas and set its size and position based on the available client area.
2. Set the background colour of the `MainForm`.
3. Set the background colour and foreground colour of the drawing canvas.
4. Set the line width for the drawing canvas.
5. Clear the drawing canvas.
6. Create a `Point` object, to store the touch start position when drawing.
7. Register the `ITouchEventListener` implemented by the `MainForm` class.

The following code snippet is an example of code that would be added to the `MainForm`'s `OnInitializing()` method, in order to create a graphics canvas and enable the handling of touch events:

```
// Create a member canvas object, and set its size to the
// Form's client area
__pDrawingCanvas = new Canvas;
__pDrawingCanvas->Construct(GetClientAreaBounds());

// Set background colour of the Form
SetBackgroundColor(Color::COLOR_WHITE);

// Set the bg, fg colour and line width of canvas
__pDrawingCanvas->SetBackgroundColor(Color::COLOR_WHITE);
__pDrawingCanvas->SetForegroundColor(Color::COLOR_RED);
__pDrawingCanvas->SetLineWidth(3);
__pDrawingCanvas->Clear();

// Create an instance of a Point to store the touch start position
__pStartPoint = new Point();

// Register touch event listener
AddTouchListener(*this);
```

Because this recipe is using the touch events to create a custom drawing on a graphics canvas, the `OnDraw()` method must be implemented. The `GetCanvasN()` method is used within the `OnDraw()` implementation to retrieve the canvas the `MainForm` is using, otherwise the default `Draw()` method of the `MainForm` will overwrite the custom drawing.

In this recipe the `OnDraw()` method copies the contents of the drawing canvas to the window canvas; this is necessary because the drawing operations are not actually being performed directly on the `MainForm`'s canvas in the `OnDraw()` method:

```
result
MainForm::OnDraw(void)
{
    result r = E_SUCCESS;

    // Retrieve the window canvas
    Canvas* pWindowCanvas = GetCanvasN();

    if (pWindowCanvas)
    {
        // Copy the contents of the drawing canvas (where the drawing
        // operations have been performed) to the window canvas
        pWindowCanvas->Copy(Point(GetClientAreaBounds().x,
            GetClientAreaBounds().y),
            *__pDrawingCanvas, GetClientAreaBounds());
        delete pWindowCanvas;
    }

    return r;
}
```

The display will be updated when the `MainForm`'s `Show()` method is called; this occurs automatically following the `OnDraw()` callback.

In order to handle the touch events when they occur the following methods have to be implemented by the `MainForm`: `OnTouchDoublePressed()`, `OnTouchFocusIn()`, `OnTouchFocusOut()`, `OnTouchLongPressed()`, `OnTouchLongPressed()`, `OnTouchMoved()`, `OnTouchPressed()`, or `OnTouchReleased()`. Each method is passed information about the source of the event, the current position, the start position and the status of the event.

In this recipe only the `OnTouchPressed()`, `OnTouchMoved()` and `OnTouchDoublePressed()` methods have been implemented.

The `TOUCH_PRESSED` event occurs when the user touches the screen. This event is handled in the `MainForm` by the `OnTouchPressed()` method.

In this recipe the `OnTouchPressed()` method is used to set the touch start position `Point` object to the current position touched by the user:

```
void
MainForm::OnTouchPressed(const Osp::Ui::Control& source,
    const Osp::Graphics::Point& currentPosition, const Osp::Ui::TouchEventInfo &
touchInfo)
{
    // Set the touch start position to the current position
    __pStartPoint->SetPosition(currentPosition);
}
```

This touch start position is used in the `OnTouchMoved()` method for the line drawing.

The `TOUCH_MOVED` event occurs when the user is touching the screen and they have moved their finger across the screen. The first event is sent after the user has moved 22 pixels in either the x or y direction, and thereafter events are sent for every movement of 2 pixels or more in either direction. This event is handled in the `MainForm` by the `OnTouchMoved()` method.

In this recipe the `OnTouchMoved()` method is used to draw a line from the stored touch start position to the current position:

```
void
MainForm::OnTouchMoved(const Osp::Ui::Control& source,
    const Osp::Graphics::Point& currentPosition, const Osp::Ui::TouchEventInfo &
touchInfo)
{
    // Draw a line from stored touch start point to current position
    __pDrawingCanvas->DrawLine(*__pStartPoint, currentPosition);
}
```

```
// Draw the Form
RequestRedraw();

// Set the current position to the touch start point
__pStartPoint->SetPosition(currentPosition);
}
```

After the line is drawn, the touch start position is set to the current position. This is necessary to draw a line that correctly represents the user's motion across the screen. The start position passed into this function cannot be used because it is the start position of when the user touched the screen, not the start position of the last movement.

The `TOUCH_DOUBLE_PRESSED` event occurs when the user double taps on the screen. This event is handled in the `MainForm` by the `OnTouchDoublePressed()` method.

In this recipe the `OnTouchDoublePressed()` method is used to clear the graphics canvas:

```
void
MainForm::OnTouchDoublePressed(const Osp::Ui::Control& source,
    const Osp::Graphics::Point& currentPosition, const Osp::Ui::TouchEventInfo &
touchInfo)
{
    // Clear the canvas
    __pDrawingCanvas->Clear();

    // Draw the Form
    RequestRedraw();
}
```

## Hints, pitfalls and related topics

As would be expected, the `TOUCH_PRESSED` and `TOUCH_RELEASED` events occur in pairs when the user touches the screen. These events also occur in conjunction with the `TOUCH_MOVED`,

TOUCH\_LONG\_PRESSED and TOUCH\_DOUBLE\_PRESSED events. These event sequences need to be taken into account when designing an application's event processing.

The sequence of events for a movement touch event is:

1. TOUCH\_PRESSED
2. TOUCH\_MOVED (multiple times)
3. TOUCH\_RELEASED

The sequence of events for a long press touch event is:

1. TOUCH\_PRESSED
2. TOUCH\_LONG\_PRESSED
3. TOUCH\_RELEASED

The sequence of events for a double-tap touch event is:

1. TOUCH\_PRESSED
2. TOUCH\_RELEASED
3. TOUCH\_DOUBLE\_PRESSED
4. TOUCH\_RELEASED

It might be suitable for an application to only process a TOUCH\_PRESSED event when its corresponding TOUCH\_RELEASED event has been received. Or alternatively, the application may be designed so that any processing done in the `OnTouchPressed()` method is reversed if the TOUCH\_MOVED, TOUCH\_LONG\_PRESSED or TOUCH\_DOUBLE\_PRESSED event is received in the sequence afterwards. In both these cases it would be necessary to store the status of the previous event(s).

It is advisable not to do too much processing in the `OnTouchMoved()` method, as the sensitivity (2 pixels) of movement can result in this method being called many times.

In this recipe, the `GetCanvasN()` method is used in the `OnDraw()` method to retrieve the `MainForm`'s canvas. To prevent memory leaks, if a method has an 'N' postfix, the caller must delete the returned instance after the caller is finished with the instance.

## **Related recipes**

### 2.1 Adding a Form recipe

---

## 2.8 Using base applications

### Problem description

You want to launch a base application from a bada application using the Application Control.

### The recipe

The following configuration is required for this recipe:

**Namespaces used:**

Osp::Ui, Osp::Ui::Controls, Osp::App, Osp::Base::Collection

**Header files to #include:**

FUi.h, FUiControls.h, FApp.h, FBase.h

**Required libraries:**

FUi, FUiControls, FApp, FBase [.lib | .so]

**Required privilege level:**

Normal

**Required privilege groups:**

WEB\_SERVICE<sup>4</sup>

Base applications are those which are stored in the built-in memory of the device and are not removable via the Application Manager. Base applications export features which are commonly required by other applications e.g. playing a video, or retrieving a contact. Access to these features is provided through Application Control.

The Application Manager is used to find the base application that provides the feature required by an application, and an instance of the Application Control is returned. The Application Control is then used to start a base application and control specific behavior. Once the Application Control is started, the application goes to the background and the target application control UI is displayed.

---

<sup>4</sup> Required for example 1 only.

This recipe shows two different examples of accessing functionality from a base application. Each example details the steps required to use the Application Control to access the necessary functionality, and provides some sample code snippets.

Both examples in this recipe assume the steps detailed in method 2 of the ‘Adding a Form’ recipe (2.1) have been followed to add a Form to the Application’s Frame.

### Example 1: Launching the browser application

This recipe uses Application Control to launch the base application Browser with a specific URI. The relevant data to be delivered to the Application control, in this case the URI, is constructed using a collection of type `Osp::Base::Collection::ArrayList`.

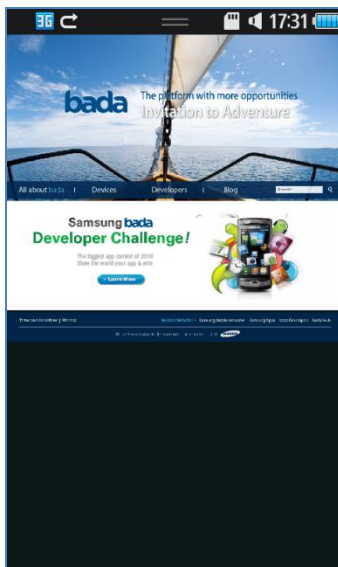


Figure 2.8.1 Application Control – launching the browser

As the Browser Application control does not return any status information or data, no callback listener is required.

The `WEB_SERVICE` group privilege is required to use the Browser Application control.

The following steps are necessary to launch the Browser base application with the specified URI:

1. Use the Application Manager to find the Application Control for launching the Browser.
2. Create an instance of an ArrayList collection to store the application control data in.
3. Create an instance of a String and initialise it with the “url:” tag and the URI for the Browser Application control to launch.
4. Start the Browser Application control, passing in the URI data in an ArrayList.

The following code snippet is an example of a method that would be added to the MainForm’s class, in order to launch the browser to a specified URI:

```
result
MainForm::LaunchBrowser(void)
{
    result r = E_SUCCESS;

    // Find the required Application Control based on the application
    // control id and the type of operation required
    AppControl* pAc = AppManager::FindAppControlN(APPCONTROL_BROWSER,
                                                OPERATION_MAIN);

    if(pAc != null)
    {
        // Create an ArrayList and initialise
        ArrayList* pDataList = null;
        pDataList = new ArrayList();
        pDataList->Construct();

        // Create String for URI and add to the ArrayList
        String* pData = null;
        pData = new String(L"url:http://www.bada.com");
        pDataList->Add(*pData);

        // Start the required application with the relevant data.
        // As the BROWSER does not provide any callback data, no listener is required.
        r = pAc->Start(pDataList, null);

        delete pAc;
    }
}
```

```
pDataList->RemoveAll(true);
delete pDataList;
}
else
{
    r = E_OBJ_NOT_FOUND;
}
return r;
}
```

Once the Application control `Start()` method has been called the application goes to the background and the Browser UI is displayed. When the user exits the Browser application, the application returns to the foreground.

### Example 2: Retrieving phone numbers from the contacts list

This recipe uses Application Control to launch the Contact application and allows the user to select multiple phone numbers from the list of contacts. The relevant data to be delivered to the Application control, in this case the selection mode and the return type, is constructed using a collection of type `Osp::Base::Collection::ArrayList`.

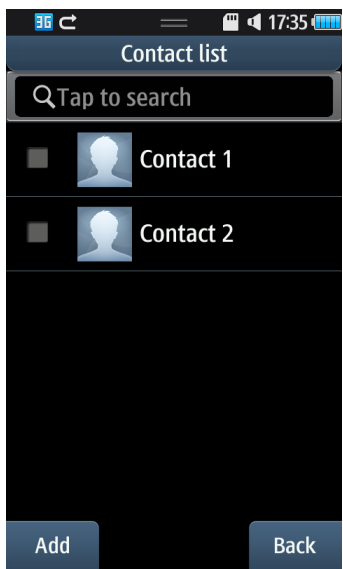


Figure 2.8.2 Application Control - select items from Contact application

In order to allow the user to select phone numbers from the Contact application it is launched with the PICK operation. This operation results in some data being returned from the Contact Application Control when it is exited. This data is the result of the operation and the value strings of the selected contacts. In order to receive data from the Contacts Application control the application needs to implement the `Osp::App::IAppControlEventsListener` interface.

The following code snippet shows an example of the `MainForm` class definition:

```
class MainForm :
    public Osp::Ui::Controls::Form,
    public Osp::App::IAppControlEventsListener
{
// Construction
public:
    MainForm(void);
    ~MainForm(void);
    bool Initialize(void);

public:
    result OnInitializing(void);
    result OnTerminating(void);

    void OnAppControlCompleted(const Osp::Base::String& appControlId,
        const Osp::Base::String& operationId,
        const Osp::Base::Collection::IList* pResultList);

    result GetContact(void);

    void DisplayContact(String* number);
};
```

The following steps are necessary to launch the Contact base application:

1. Use the Application Manager to find the Application Control for launching the Contact application with the PICK operation.
2. Create an instance of an `ArrayList` collection to store the application control data.
3. Create an instance of a `String`, initialize it with the “selectionMode:” tag and the selection mode for the Contact application (in this example multiple selection mode is defined) and add it to the `ArrayList`.

4. Create an instance of a `String`, initialise it with the “returnType:” tag and the return type for the Contact application (in this example the phone number value string is requested) and add it to the `ArrayList`.
5. Start the Contact Application control, passing in the application control data in an `ArrayList` and a pointer to the Application Control callback listener.

The following code snippet is an example of a method that would be added to the `MainForm`'s class, in order to launch the Contact application for the required operation:

```
result
MainForm::GetContact(void)
{
    result r = E_SUCCESS;

    // Find the required Application Control based on the application
    // control id and the type of operation required
    AppControl* pAc = AppManager::FindAppControlN(APPCONTROL_CONTACT,
                                                OPERATION_PICK);

    if(pAc != null)
    {
        // Create an ArrayList and initialise
        ArrayList* pDataList = null;
        pDataList = new ArrayList();
        pDataList->Construct();

        // Create String for selection mode and add to the
        // ArrayList
        String* pData1 = null;
        pData1 = new String(L"selectionMode:multiple");
        pDataList->Add(*pData1);

        // Create String for return type and add to the ArrayList
        String* pData2 = null;
        pData2 = new String(L"returnType:phone");
        pDataList->Add(*pData2);

        // Start the required application with the relevant data
        // As the CONTACTS app is going to return phone numbers,
        // the callback listener must be sent to the
```

```
// application control.
r = pAc->Start(pDataList, this);

delete pAc;

pDataList->RemoveAll(true);
delete pDataList;
}
else
{
    r = E_OBJ_NOT_FOUND;
}

return r;
}
```

The `IAppControlEvents` interface receives notification events from the Application Control when it is done (i.e. the operation has completed or has been cancelled). The type of notification data received depends on the type of Application control started and the operation type used. For this example, the implementation of the `OnAppControlCompleted()` method must handle both Application status results and multiple items of Contact data.

The following steps describe the functionality that would be implemented in the `OnAppControlCompleted()` method:

1. Retrieve the number of result elements in the `IList` collection returned from the Contact application.
2. Check the application control ID and the operation ID match those requested when the application control was started (`APPCONTROL_CONTACT` and `OPERATION_PICK` respectively).
3. If they match, determine if the request was successful by retrieving the first result item from the `IList` collection and comparing it with the `APPCONTROL_RESULT_SUCCEEDED` result.
4. If it matches, determine if the correct return type has been set by retrieving the second result item from the `IList` collection and comparing it with the string "phone".
5. If it matches, retrieve all the data values by looping through the list until the last element is reached.

The following code snippet is an example of the Application Control event handling in the `MainForm`'s `OnAppControlCompleted()` method:

```
void
MainForm::OnAppControlCompleted(const Osp::Base::String& appControlId,
    const Osp::Base::String& operationId,
    const Osp::Base::Collection::IList* pResultList)
{
    String* pPickResult = null;

    int loopCount = 0;
    int elementCount = pResultList->GetCount();

    if(appControlId.Equals(APPCONTROL_CONTACT) &&
        operationId.Equals(OPERATION_PICK))
    {
        pPickResult = (String*)pResultList->GetAt(0);
        if(pPickResult->Equals(APPCONTROL_RESULT_SUCCEEDED))
        {
            pPickResult = (String*)pResultList->GetAt(1);
            if(pPickResult->Equals(String(L"phone")))
            {
                // Result is a phone number,
                // so retrieve numbers from the list
                for (loopCount=2; loopCount<elementCount; loopCount++)
                {
                    pPickResult = (String*)pResultList->GetAt(loopCount);
                    DisplayContact(pPickResult);
                }
            }
        }
    }
}
```

In this example a `DisplayContact()` method can be implemented to display the phone numbers retrieved from the Contact application.

Once the Application control `start` method has been called the application goes to the background and the Contact UI is displayed. When the user selects required items or exits the Contact application, the application returns to the foreground.

## **Hints, pitfalls and related topics**

The `WEB_SERVICE` group privilege must be set in the Application's manifest file in order to use the Browser Application control.

When using the Contact application control it is not possible to request the return of multiple items of data related to a single contact (e.g. both the phone number and the Email address). If this functionality is required the following steps should be performed instead:

1. Use the Contact application control for the user to select the required contact, but request the return type from the operation to be the `contact ID`.
2. Use the `Osp::Social::Addressbook` class's `GetContactN()` method to retrieve all the contact details associated with the contact.

The `ADDRESSBOOK` group privilege must be set in the Application's manifest file in order to use this Address book functionality.

When selecting multiple items from the Contact application, they may not be added to the `IList` collection in the same order as they are presented in the Contact application. They will be returned in the order of ascending contact ID (i.e. in the order in which the information was saved in the Contact application).

## **Related recipes**

### 2.1 Adding a Form recipe

---

## 2.9 Get magnetic field from Compass Sensor

### Problem description

You want to obtain data from the magnetometer (compass sensor) to interpret that information for various applications (e.g., determine device orientation or metal detection).

### The recipe

The following configuration is required.

**Namespaces used:**

`Osp::Uix`

**Header files to #include:**

`FUixSensorManager.h`, `FUixSensorTypes.h`

**Required libraries:**

`FUix[.lib | .so]`

**Required privilege level:**

Normal

**Required privilege groups:**

none.

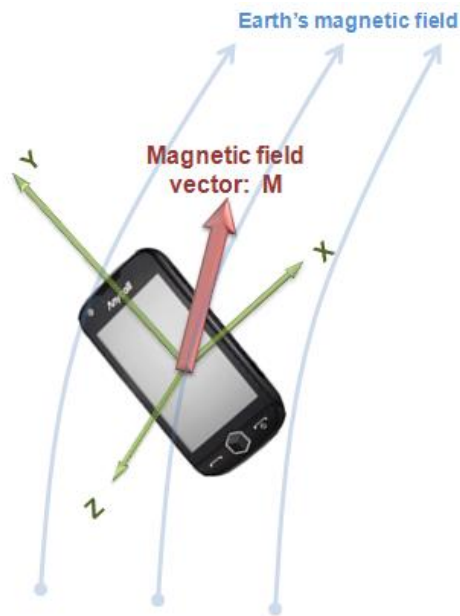
This recipe is related to the namespace `Osp::Uix`. `Uix` stands for UI extension, which provides additional novel ways for the user to interface with a device. These are not covered in basic UIs.

Traditional UI approaches mostly depend on oral or visual in- and output for providing information, or they take commands or information from the user through buttons. UI extension is intended to improve the user's experience in interacting with a mobile phone.

This recipe covers one particular part of that namespace. We make use of the magnetometer (magnetic sensor). This sensor is a 3-axis electronic compass. It can also be used for determining the orientation. The magnetic sensor measures the magnetic field strength. The measurements are split into the X, Y, and Z axis and the unit is Micro-Tesla ( $\mu\text{T}$ ). Note that several factors can have an impact on the data coming

from this sensor: weather, your current location on the planet, and most influentially nearby magnetic fields, such as magnets or electric coils.

The magnetic sensor uses the phone as a fixed frame of reference as indicated in the figure below and models this into the 3-axis Cartesian space coordinate system.



**Figure 2.9.1 Magnetic field related to the device**

For this example we use the `SensorManager` class and the `ISensorEventListener` interface both from the namespace `OSP: :Uix`. The `SensorManager` class provides the functionality for adding and removing sensor listeners, checking if a sensor is available, setting intervals and getting the max and min intervals. A bada device can have various physical sensors (for example, an acceleration sensor, GPS sensor, or proximity). The `SensorManager` supports simple access of built-in sensors in a device. By registering a listener to the `SensorManager`, an application can get sensor data with a timestamp based on definable intervals.

In our example we define a class that inherits from the `ISensorEventListener` interface. Below you can find a sample header file that contains all the signatures of the methods we will show in this recipe.

```
//...

class YourClassName :
    public Osp::Uix::ISensorEventListener
{
private:
    Osp::Uix::SensorManager __sensorMgr;

    // ...

public:
    // ...
    bool RegisterSensor();
    bool UnRegisterSensor();
    void OnDataReceived(Osp::Uix::SensorType sensorType,
        Osp::Uix::SensorData& sensorData, result r);

private:
    void DrawArrow();
    float GetDegree(float rad);
    float CalculateAngle(float f1, float f2);

};
```

Because we inherit from an interface, we need to overwrite the respective method with the following signature:

```
void OnDataReceived(Osp::Uix::SensorType sensorType,
    Osp::Uix::SensorData& sensorData, result r);
```

The implementation of our class also provides two methods that help to register and unregister a sensor type.

```
bool
RegisterSensor()
{
```

```
result r = E_SUCCESS;

// Construct the sensor manager instance
r = __sensorMgr.Construct();
if(IsFailed(r))
{
    return false;
}

long interval = 0;

// Activate the sensor type SENSOR_TYPE_MAGNETIC
r = __sensorMgr.GetMinInterval(SENSOR_TYPE_MAGNETIC, interval);
if(IsFailed(r))
{
    return false;
}
if (interval < 50)
{
    interval = 50;
}
// Register our listener to this sensor
r = __sensorMgr.AddSensorListener(*this, SENSOR_TYPE_MAGNETIC,
                                   interval, false);

if(IsFailed(r))
{
    return false;
}

return true;
}
```

```
bool
UnRegisterSensor()
{
    result r = E_SUCCESS;

    // Deactivate our compass sensor
```

```
r = __sensorMgr.RemoveSensorListener(*this);
if(IsFailed(r))
{
    return false;
}
return true;
}
```

In the body of the overwritten `OnDataReceived()` method we get the X, Y, and Z values and can process them as needed. In this example we simply calculate the 3D vector of the earth's magnetic field (assuming that there are no other sources of severe inferences). For this purpose we defined a member of type float called `__magField` into which we store the length of this vector.

```
Void
OnDataReceived(Osp::Uix::SensorType sensorType, Osp::Uix::SensorData&
    sensorData, result r)
{
    float x=0 ,y=0 ,z=0;

    AppLog("DataReceived: Get Magnetic Sensor Data");
    sensorData.GetValue((SensorDataKey)MAGNETIC_DATA_KEY_X, x);
    sensorData.GetValue((SensorDataKey)MAGNETIC_DATA_KEY_Y, y);
    sensorData.GetValue((SensorDataKey)MAGNETIC_DATA_KEY_Z, z);

    // Calculate the length of our vector determined by the three axis
    __magField = Math::Sqrt((Math::Pow(x,2)+ Math::Pow(y,2)+ Math::Pow(z,2)));
}
```

## Hints, pitfalls, and related topics

Once your code compiles without errors you can run it in the simulator. A very convenient feature of the bada simulator is the Event Injector tool. To get this, right-click on the simulator screen, select the Event Injector, and there choose the Sensors tab. In the row below choose the Magnetic tab. There you can

freely choose the  $\mu T$  values per X, Y, and Z axis. By clicking on the Send button these are then sent as events to your application running on the bada simulator. This tool is very useful for testing.

For some use cases you might be interested in getting the angle between two vectors. The following code snippet should help.

```
float
CalculateAngle(float f1, float f2)
{
    float angle = 0;

    // angle in rad
    if (f1==0 && f2==0){
        angle = 0;
        return angle;
    }else{
        angle = Math::Asin(f1 / Math::Sqrt((Math::Pow(f1,2) +
            Math::Pow(f2,2))));
    }
    return angle;
}
```

The SDK calculates the angle in rad. For converting this into degrees you may find the following code useful.

```
float
GetDegree(float rad)
{
    return Math::Abs(rad * 180/Math::GetPi());
}
```

---

## 2.10 Get geographic data from provider and show map

### Problem description

You want to obtain geographic information and display it in form of a map on your bada device. In addition you want to show the current position of a user on that map.

### The recipe

The following configuration is required.

**Namespaces used:**

```
Osp::Locations, Osp::Locations::Services, Osp::Locations::Controls,  
Osp::Graphics
```

**Header files to #include:**

```
FLocations.h
```

**Required libraries:**

```
FLocations, FLocationControls[.lib | .so]
```

**Required privilege level:**

```
Normal
```

**Required privilege groups:**

```
LOCATION, LOCATION_SERVICE
```

The namespace `Osp::Locations` contains among others the classes `LocationProvider` and `Location`, which we will use in our recipe. With the `LocationProvider` class you can define where you want to get a position from (e.g. from the GPS sensor). This class also implements the interface `ILocationListener`, which gives you information about location updates. The class `Location` representing the location information is a part of `LocationProvider`.

The namespace `Osp::Locations::Services` provides a number of useful services that can be exploited by the developer. An essential service that we will also introduce is what is encapsulated in the `IMapServiceProvider` interface.

Finally, the last namespace presented here is `Osp::Locations::Controls`, which allows users to pan, zoom and interact with maps as is generally expected.

We simply show how a map can be displayed on screen that has an overlay additionally showing the users current position.

Our sample code presents snips from two C++ source files and their corresponding header files:

- `MapVisualisation.cpp` (covers code for presenting geographic data as a map)
- `MapInteraction.cpp` (covers code for obtaining locations)

In order to be able to display maps we need access to a geographic data provider. For this example we use deCarta as our map provider. You can sign up at deCarta (<http://developer.decarta.com>) and request a key for free.

Once you have your key, you can link your deCarta account in the application by creating a map provider object. We do this in our `MapVisualisation` class.

```
// Define a name for the provider
String providerName = L"deCarta";

// Set a map key for deCarta (insert your username/pwd instead of XXX/YYYY)
static const String extraInfo = L"ClientName=XXX;ClientPassword=YYY;"
    "HostUrl=http://developer.kr.dz.decarta.com:80/opensls/opensls";

// Create a map provider
IMapServiceProvider* pProvider = static_cast<IMapServiceProvider*>(
    ProviderManager::ConnectToServiceProviderN(L"deCarta",
        LOC_SVC_PROVIDER_TYPE_MAP, extraInfo));
```

After that, construct a `Map` control and configure it according to your display requirements.

```
// Create Map control
Map* pMap = new Map();

// Construct the map
pMap->Construct(Osp::Graphics::Rectangle(0, 0, GetCanvasN()->GetBounds().width,
    GetCanvasN()->GetBounds().height), *pProvider);

// Configuration: allow panning
pMap->SetPanEnabled(true);

// Configuration: define the center of the map
// These are the coordinates of Staines, UK, around which we center the map
double lat = 51.433315; double lon = -0.497382;
pMap->SetCenter(lat, lon, false);

// Configuration: Show my location
pMap->SetMyLocationEnabled(true);
```

It is important to set the configuration value for the selected map area. The default value is 'global-mobile'. Other area codes can be found on the deCarta website. For demonstration purposes we will use this string to access global map data. This is done by invoking `SetPreferencePropertyValue()` method which is a member of the map class. Further map configuration settings can be obtained from the deCarta developer site.

```
// Set configuration values
static const String CONFIGURATION_VALUE(L"global-mobile");

// Configuration: Set the maps to this area
pMap->SetPreferencePropertyValue(String(L"configuration"),
    &CONFIGURATION_VALUE);
```

Now we switch to the `MapInteraction` class. Create and construct a `LocationProvider` by adding in your location source. We're using the GPS receiver. Here you can also configure the update rate, which we set to 5 seconds in the `RequestLocationUpdate()` method.

```
// Create a LocationProvider
LocationProvider *locProvider = new LocationProvider();

// Construct the LocationProvider by using GPS as its locating mechanism
locProvider->Construct(LOC_METHOD_GPS);

// Configure the time interval for updates
locProvider->RequestLocationUpdates(*this, 5, false);
```

In order to receive location updates you need to override the `OnLocationUpdated()` method which is declared in the `ILocationListener` interface.

```
//Called when a location event is received
void MapInteraction::OnLocationUpdated(Location& location){

    if(location.GetQualifiedCoordinates()!=null)
    {
        const QualifiedCoordinates *q = location.GetQualifiedCoordinates();

        // The lat and lon values
        double longitude = q->GetLongitude();
        double latitude = q->GetLatitude();

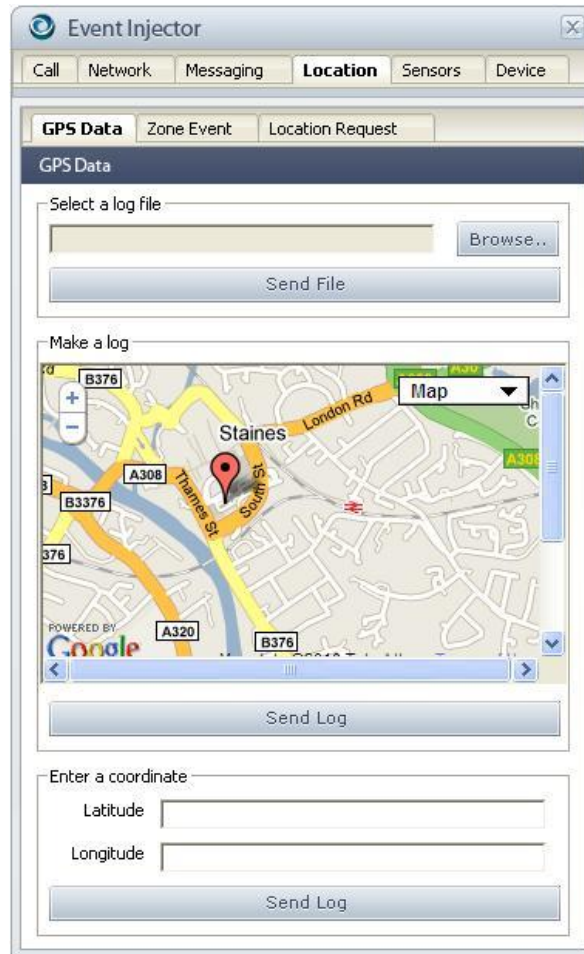
        // Center the map around these new values
        pFormMap->SetCenter(latitude, longitude, true);

        // Redraw and show the map
        pFormMap->Draw();
        pFormMap->Show();
    }
}
```

## Hints, pitfalls, and related topics

Once your code compiles without errors you can run it in the emulator. A very convenient feature of the bada emulator is the Event Injector tool. To get this, right-click on the emulator screen, select Event Injector, and choose the Location tab. You can drag and drop the pointer on the map, or directly enter

latitude and longitude values. These are then sent as events to your application running on the bada emulator. This tool is very useful for testing.



**Figure 2.10.1 Event Injector for simulating location changes**

If no map is shown on the emulator screen, you might first check whether you have any spelling errors in username and password which you got from your geographic data provider. Another reason might be that you are in a proxy environment. Then you have to set the proxy information in the emulator. Go to menu – settings – connectivity – network – bada.

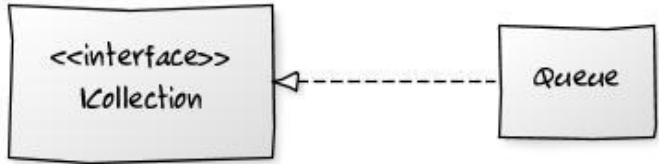
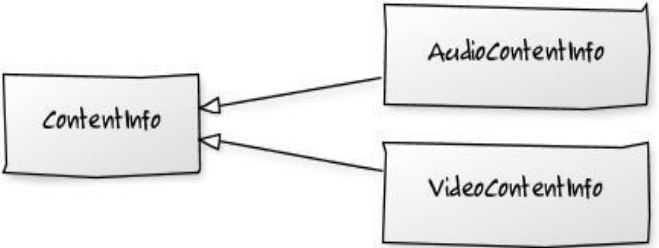
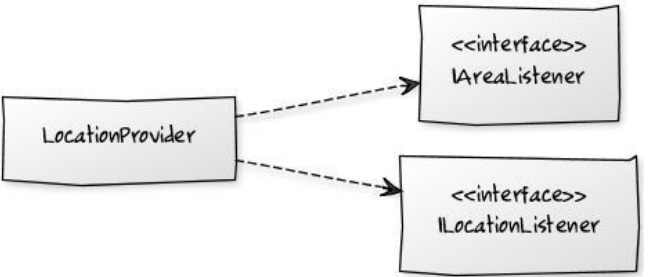
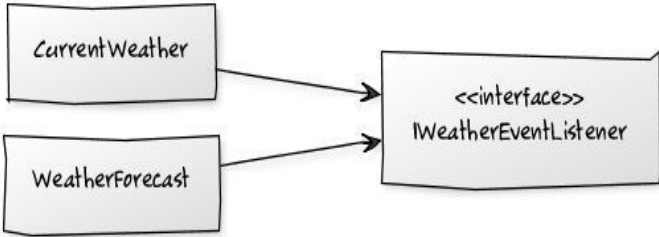
## **Example use in BuddyFix**

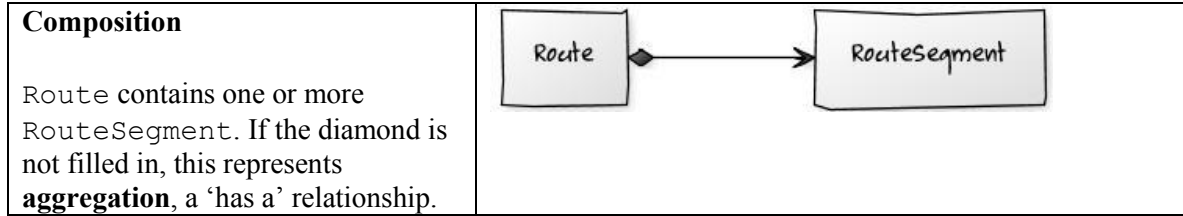
In BuddyFix we used this recipe to connect to deCarta as our geographic data provider, displayed a map and on top of the map indicated the current position of our Buddies.

# Appendix A: A UML primer

Here's an explanation of the subset of UML notation that we've used in Chapter 6:

## 1. Class diagrams

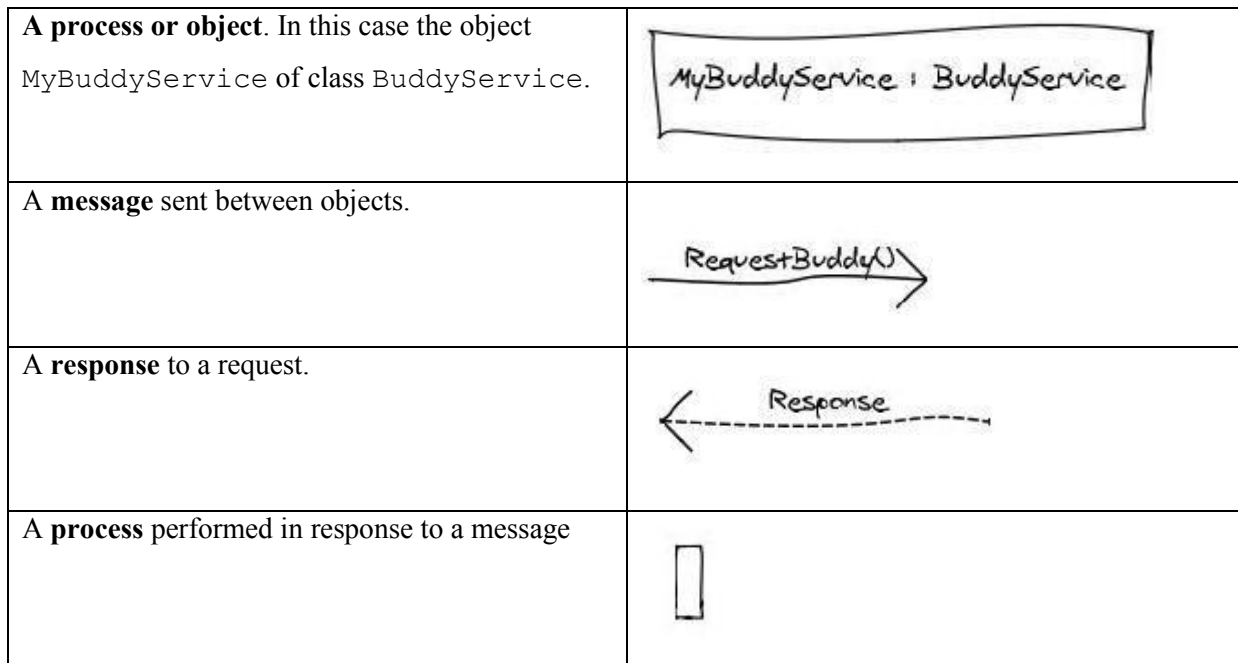
<p><b>Interface inheritance</b></p> <p>Queue derives from and implements the interface defined by ICollection. See Chapter 4, "bada fundamentals" for more information about interfaces.</p>	 <pre> classDiagram     class ICollection {         &lt;&lt;interface&gt;&gt;     }     class Queue     Queue .. &gt; ICollection             </pre>
<p><b>Inheritance</b></p> <p>Both AudioContentInfo and VideoContentInfo derive from ContentInfo</p>	 <pre> classDiagram     class ContentInfo     class AudioContentInfo     class VideoContentInfo     ContentInfo &lt; -- AudioContentInfo     ContentInfo &lt; -- VideoContentInfo             </pre>
<p><b>Dependencies</b></p> <p>LocationProvider uses IAreaListener and ILocationListener to provide location updates.</p>	 <pre> classDiagram     class LocationProvider     class IAreaListener {         &lt;&lt;interface&gt;&gt;     }     class ILocationListener {         &lt;&lt;interface&gt;&gt;     }     LocationProvider ..&gt; IAreaListener     LocationProvider ..&gt; ILocationListener             </pre>
<p><b>Simple Association</b></p> <p>CurrentWeather and WeatherForecast are connected to IWeatherEventListener in some way, the classes work together. The application requests a list of WeatherForecast or the CurrentWeather and this will be returned using the IWeatherEventListener.</p>	 <pre> classDiagram     class CurrentWeather     class WeatherForecast     class IWeatherEventListener {         &lt;&lt;interface&gt;&gt;     }     CurrentWeather --&gt; IWeatherEventListener     WeatherForecast --&gt; IWeatherEventListener             </pre>



**Note:** the UML class diagrams in Chapter 6 were created using the excellent free online tool yUML, available at: <http://yuml.me/>.

## 2. Sequence diagrams

Sequence diagrams show how objects and processes interact with each other over a timeline- the vertical line shown on the diagrams in Chapter 5, representing the lifetime of an object. In Chapter 5, we use sequence diagrams to show how the application interacts with the bada server in response to user requests, such as to make their profile searchable, and how messages are sent between the objects involved in handling the request.



**Note:** the sequence diagrams used in Chapter 5 were created using the free online web tool at <http://www.websequencediagrams.com/>

# Appendix B: A software engineering process model for mobile app development

As we mentioned early in the book (see Chapter 1), in order to successfully develop a mobile software solution you should follow an engineering process that helps you address the specific characteristics of mobile software. We refer to this as **mobile software engineering**. Such a process ideally is highly agile incorporating several macro and micro iterations. Of course, you can follow any process you prefer or develop your software without following any process at all. There is a range from strict *waterfall* model to *cowboy coding*. As a rule of thumb the more complex a project is and the more coordination it requires the more formalization in terms of processes is advisable. Experience showed that so called agile processes are very appropriate with software for fast paced markets, which definitely is the case in mobile app development.

From a high-level point of view, it might not make a difference if software for desktops, servers, the Web or for mobile devices is developed. It is all software engineering and the basic steps are always the same: requirements, design, programming, testing and deployment. However, it is the detail that makes the difference. From experience we can tell that it is not possible to simply transfer the techniques of traditional software engineering one-to-one to mobile software engineering without significant modifications.

Since in this book we are talking about the bada platform for mobile applications, our prime interest in this chapter, naturally, is the mobile software engineering process. In the following pages we will introduce one such agile mobile software engineering process that subsumes several recommended best-practices. Of course, you are free to choose whichever engineering process you want to follow and whether you comply with all the phases we introduce, pick out the most relevant for your needs or don't

use it at all. We will also show where and how the bada platform and its tools support the described phases and best-practices.

### **Some mobile software engineering best-practices**

For developing mobile software, we recommend deploying ideas from agile development initiatives (stated in the agile manifesto) such as adaptability, iterations, and making heavy use of prototyping and diversified testing as early as possible in the process.

Prototypes can be exploited nearly in any phase of the engineering of a mobile software solution. They are primarily helpful in eliciting requirements or to get a common understanding with various stakeholders early in the project. Testing surely is not unique to *mobile* software engineering but must be treated and executed differently as we will argue throughout this chapter.

The mobile software engineering process that we introduce here is subdivided into three major phases:

1. Feasibility and economic efficiency analysis phase
2. Software product realization phase
3. Distribution phase

The following sections discuss each of these phases in detail and the figure below gives an illustration of the whole process. The phases of this mobile software engineering process mainly interweave with the concept, market entry, development, and evolution phases of software product management, where clearly the strongest overlap can be mapped to the development phase.

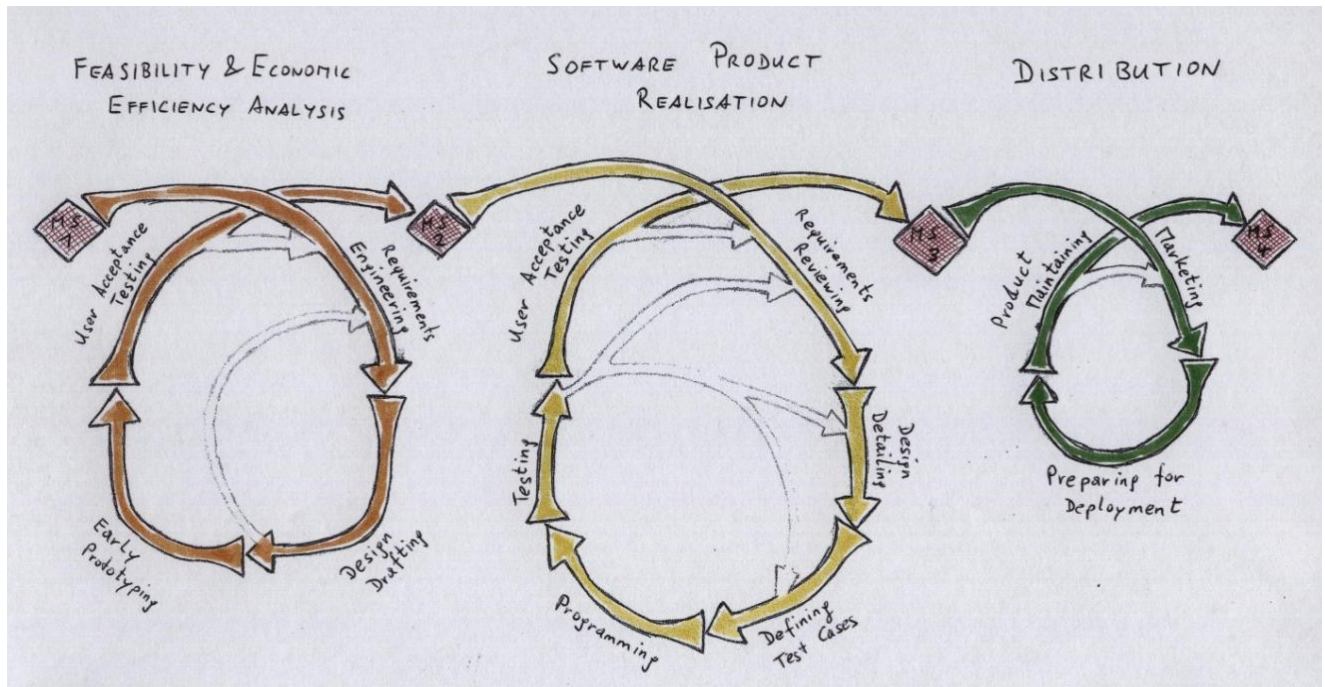


Figure A.1 A software engineering process model

### Phase 1: Feasibility and economic efficiency analysis

This first phase should help all the stakeholders to get a better picture about the feasibility, the acceptance, and the potential economic success of a mobile software solution. With economic efficiency analysis we mean that very early in the project lifetime it can be determined whether it is worth to pursue a mobile software development project. This first phase is composed of the four stages requirements engineering, design drafting, early prototyping, user acceptance testing, and a milestone at the end of the phase.

#### Stage 1.1: Requirements engineering

The first step in this analysis phase is the elicitation and understanding of the requirements. The business requirements cover considerations about the business model and distribution policy. This is input which usually originates from the product management process that embeds our mobile software engineering process.

User requirements are the starting point and represent what users need and what they expect from a mobile software. These requirements must be collected and can be maintained in form of use cases or user stories. Early prototypes can already be exploited for this elicitation because users can get a more tangible impression about the future system. Such early prototypes can be low-fidelity ones such as drawings on a paper. More advanced ones can be done by electronic images that can actually allow showing a sequence and simple interactions. Even more advanced is the use of simple UIs that can be created fairly easy and quickly by using the UI builder provided by the bada IDE. The envisioned application can be sketched and simple user interaction (clicking on buttons or exploiting multi-touch) can be modeled and even deployed and shown on a real hardware device. This helps the user even more to think of the future use of an application, which is directly reflected in the quality of the users' feedback. Chapter 2 shows the usage of the bada UI builder.

For mobile applications we recommend to focus on addressing a clearly defined problem or need with as little functionality as required. Many applications seem overloaded and as a result are difficult to use. Focus on the core and address rather less needs. We are not talking about desktop applications. We outlined the differences between mobile and desktop apps above. As mentioned the cognitive resources and the attention time span is substantially more limited. Traditional software engineers tend to underestimate the impact that a mobile setting will have on application usage patterns.

### **Stage 1.2: Design drafting**

The drafting of the design deals with two aspects. The first one being the draft of the dialog flow logic and the user interface (UI) design. This is the only part of the software that is exposed to the user and hence, the only part she can experience. From user workshops and feedback to our applications we know that it is tremendously important to provide a simple yet attractive user interface. If an application is too complex to understand within the first minute or it is ugly looking, the user will most probably not launch it a second time.

The second aspect deals with preliminary software component architecture considerations. The UI design activities are conducted in various micro iterations with feedback from stakeholders.

### **Stage 1.3: Early prototyping**

Early prototypes embody the requirements gathered and the design developed so far. They are a good means to get a common understanding. Experience showed that some rudimentary functionality should be in place such that minimal interaction is possible. That can be achieved by paper prototypes, an interactive slide show or a click-able UI mock-up. The goal is to find a good balance between a fast and cost effective prototype and providing a user experience which comes close to the final product.

As mentioned above, the bada tools give an ideal support for building good prototypes fast. A further advantage is that these prototypes can be re-used for the later programming as is without the need of entirely re-doing all the work for the UI. This is a result from the clear separation of UI design and programming brought to you by bada.

### **Stage 1.4: User acceptance testing**

This is an optional stage. However, we recommend to plan and execute tests with users that are not involved in the project. These tests can be short in length but should be as close to the real-world context as possible. This way, many problems that will go unnoticed in a lab environment will become apparent, such as bad readability under sunlight or overly complex and confusing user interfaces. These kinds of trials will almost always yield highly valuable feedback about the future acceptance by users and thus the potential success on the market. After a test session there may be one or more iterations back to requirements engineering stage depending on available time, quality and cost targets.

The first phase is ended with the milestone Decision for Continuation, which entails the decision to pursue or abandon the software project. With the knowledge gained in this phase – particularly arising from the early acceptance tests – the potential success of a mobile solution can be estimated much clearer, which helps to assess the economic efficiency.

### **Phase 2: Software product realization**

The second main phase deals with realizing the software solution. It bases on the works of the first phase where as many results as possible should be reused. To follow this agile engineering process, also this

phase is characterized by many iterations, incremental development ('first things first'), and a high degree of internal and external communication.

This phase is composed of the six stages requirements reviewing, design detailing, defining test cases, programming, testing, user acceptance testing, and a milestone at the end of the phase. Again, we would like to emphasize that we present a complete mobile software engineering. Due to our advice about working in an *agile* way, which implies adaptability, you are free of choosing which ever stages you want to deploy in your software project.

### **Stage 2.1: Requirements reviewing**

Prior to starting with the detailed software design, programming, and testing the requirements should be reviewed and revised – ideally together with all stakeholders.

### **Stage 2.2: Design detailing**

In this step the available UI and architecture designs of the first phase are detailed into much more fine-grained levels – down to element and component level. Although, mobile software engineering is a comparatively young discipline, some (mostly vendor-driven) initiatives emerged that provide guidelines about user interface design. Although you are not bound to use these, we strongly recommend complying with such guidelines because it simply increases the usability of your product and the acceptance.

Also the bada platform introduced such guidelines. These are implicitly used when you use the open APIs for the UI and extended UI controls. Apart from that the IDE ships with a help document that gives further information about best-practices for ideal UI design.

### **Stage 2.3: Defining test cases**

Testing is absolutely necessary in any software engineering endeavor. In mobile software engineering it is more multifaceted and more variables need to be taken into consideration. The definition of test cases is the first activity related to testing. The cases can be derived from the requirements.

### **Stage 2.4: Programming**

This stage means transforming the designs into program code that can finally successfully pass all test cases. Although, mobile devices get more and more powerful, we nevertheless it is highly recommendable to write efficient code with an eye on conservative computation in order to conserve as much battery capacity as possible. Stretching availability of battery power is crucial as we argued in chapter 1 earlier.

The bada IDE ideally supports you with all you need for writing your program code such as syntax highlighting, code completion, comprehensive documentation and example material, and all the necessary compiling, linking, deploying, and running functionalities.

Important is also to understand that every mobile application is subject to a ‘technical’ application lifecycle, which subsumes initializing, running, terminating, and terminated. The bada platform and its shipped IDE to access it provides the support to easily deal with this app lifecycle and to do the right things in the appropriate stages (through well-defined methods).

### **Stage 2.5: Testing**

In the mobile software engineering process, much emphasis must be placed on this stage: testing. We must differentiate between the testing platform and the real platform. The testing platform is usually a desktop computer that runs an emulation of the mobile device. Unfortunately, emulators per se exhibit a great discrepancy compared to the real mobile devices. Emulator tests are tests in ideal lab environments where context factors such as position or light conditions are either not considered or simulated. Hence, it is necessary to test software on the real target mobile device and in the real world environment where the software is intended to be used. All the “controllable” variables must be caught by the software such as behavior according to device capabilities, adapting to screen size, masking network disconnections or loss of GPS signal.

The last three stages (2.3 defining test cases, 2.4 programming, and 2.5 testing) are characterized by many micro iterations and in reality are partly also executed in parallel. Particularly testing naturally iterates with the test case definition and programming stages. It usually also iterates with the design detailing and the requirements reviewing steps.

### **Stage 2.6: User acceptance testing**

User acceptance tests at the end of the second phase are again optional but recommended. By this, engineers can make sure to really meet users' requirements with a software version that more and more approaches a final state. This type of testing can be repeated and the outcomes can be fed back into earlier steps of this phase. Human-computer interaction techniques such as audio/video recordings, questionnaires, cooperative evaluations, focus groups, or controlled experiments can be deployed and are beneficial.

After stakeholders accept the existing version of the mobile software, it can be denoted as a release. This means that the milestone Version Released is fulfilled, and the next iteration of the same phase (to realize a further increment) or the successive phase can start.

### **Phase 3: Distribution**

This phase mainly deals with bringing the mobile software product into the market to the users. This phase is less iterative than the other two. The phase is composed of the three stages marketing, preparing for deployment, product maintaining, and a milestone at the end of the phase.

#### **Stage 3.1: Marketing**

The marketing stage exemplifies where and how the processes of mobile software engineering and product management overlap and complement one another. This stage serves for the finalization of the business model and distribution policy and for preparation of the actual market entry of the software product. Marketing is partly in parallel to other steps and already starts during earlier phases as suggested by the product management process.

#### **Stage 3.1: Preparing for deployment**

In this step, the software must be prepared (i.e. signed) such that the needed functionality of a mobile device can be accessed appropriately. It must be decided which components and contents are packaged together and which parts may be downloaded or installed on-demand. Also the distribution channel must be prepared which usually involves requesting an account and getting familiar with the according policies. Finally, the mobile software is physically distributed and installed on the end users' devices.

#### **Stage 3.1: Product maintainance**

Maintenance covers support, bug fixing, and feedback integration. This step – depending on the significance of the reported issue and the policy of the application provider – can lead to an additional iteration with the preparing-for-deployment step. Also the level of maintenance and the commitment to it depends on the provider and can range from 24/7 service to no support at all.

The mobile software engineering process ends with the end event (milestone: Project End) which was defined by the stakeholders and the project manager.