MetaEdit+

Version 5.6

The Mobile UI Example

MetaCase Document No. MUE-5.6

First Printing, 1st Edition, October 2025

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä
Finland

E-mail: info@metacase.com
WWW: https://www.metacase.com

Preface

The Mobile UI example illustrates how cellular phone applications can be modeled and generated using Domain-Specific Modeling (DSM). To achieve this, a domain-specific modeling language for S60 phones is implemented in MetaEdit+ along with a generator for producing code. Using the modeling language, a developer can design phone applications by directly using the domain concepts of the phone, such as its widgets and services. Generators are used to produce the executable code, automate the deployment to run the code in an emulator (with a single click), check the models and produce documentation.

This example covers both the issues related to S60 application modeling as well as in part how the DSM solution was made. First, we take a look at the modeling language with some examples, and then we discuss the issues of modeling language and generator specification. We will also discuss ports, and show how to import external symbol elements in order to define notation. On the code generation side, special focus will be given to integrating library code and manually written code with code generated from models.

To explore the S60 phone examples thoroughly, the following things are required:

- MetaEdit+ for trying out the S60 phone language. The phone example can be found from the demo repository, from the project named ‘Mobile UI’. For further information about MetaEdit+, please refer to the MetaEdit+ User’s Guide.
- Nokia S60 SDK emulator for running the generated applications, and S60 Python interpreter for running Python code in the emulator. Installation instructions can be found from https://developer.nokia.com/community/wiki/Category:PySymbian. This example has been developed using S60 SDK version “2nd Ed, FP2 146 268”, but there are also newer versions available. Alternatively you can use a real target device, a Symbian smartphone using the S60 UI framework, with the S60 Python interpreter installed. We recommend that you install all programs using the default directories as suggested by the installers.

We expect that you have basic knowledge about using MetaEdit+. If you want to extend the DSM solution further – to add notational symbols, additional constraints, or extra generators, or to modify dialogs and toolbars for the modeling tool − you should have MetaEdit+ Workbench or the evaluation version available from https://www.metacase.com.

1 The S60 phone example

The S60 phone example presents a DSM language and its tool support, specifically tailored for developing applications into smartphones. The target for the code generation is Python for Series 60, a framework which runs on Symbian smartphones. Because of this target, the modeling language is based on the architecture, phone services and widgets that this particular framework provides. Naturally, it would also be possible to model and generate code for other smartphone frameworks in a similar way. As it happens, S60 supports two other frameworks and programming languages (C++ and Java) in addition to Python. A DSM solution for C++ is also implemented, in the same MetaEdit+ project, but this document will concentrate on the Python implementation.

In this first chapter we introduce the S60 modeling language and its usage scenarios. Chapter 2 goes on to explain how to use the language with an example: we modify an application design and generate code to run the modified application. Chapter 3 describes how the language and generator were defined. Rather than focusing on the basic metamodeling capabilities that are discussed in the tutorial examples on metamodeling, here we inspect selected aspects of language and generation creation. On the language side we show how to define ports for representational purposes and how to use external graphics for the notation. On the code generation side, we show different ways to refer to manually written code from models.

1.1 THE BASIC IDEA OF DSM FOR S60 PHONES

The general objective of the "S60 phone language" is to ease and speed up application development. This is achieved by raising the level of abstraction from programming concepts to user-visible UI concepts and phone services. Doing this hides the unnecessary complexity as application developers do not need to master the details of the phone architecture and related programming model. Briefly, using the modeling language a developer draws a graphical design of the application logic using S60's UI elements (lists, forms etc.) and services (SMS, accessing files, taking a picture etc). At any stage of the design phase, a developer can run a generator that produces application code ready for execution. The generated code uses the API services provided by the Python for S60 framework.

This domain-specific language propels application design toward maximum simplicity and easiness by using high-level UI concepts, widget visualization that maps 1:1 with the actual phone, and by showing behavioral logic and flow visually. Furthermore, the modeling language covers architectural rules that prevent developers from making illegal designs: ideally, if a developer can draw the application, it will work.

1.2 AN EXAMPLE

Using DSM for S60 phones, the design process goes as follows: A developer specifies the application by thinking about the relevant services of the phone, the UI that is needed and navigation flows within the application. These phone-specific concepts are directly the modeling language's concepts too. They can be selected from the editor toolbar and placed on the drawing area. The modeling elements can be further specified with related properties and connected together to specify the navigation flows.

An example of an application design is illustrated in Figure 1-1. If you are familiar with some phone applications, like phone book or calendar, you most likely already understood what the application does. It allows a phone user to register for a conference via text messaging, view the conference program and speaker data or browse the conference program via the web.

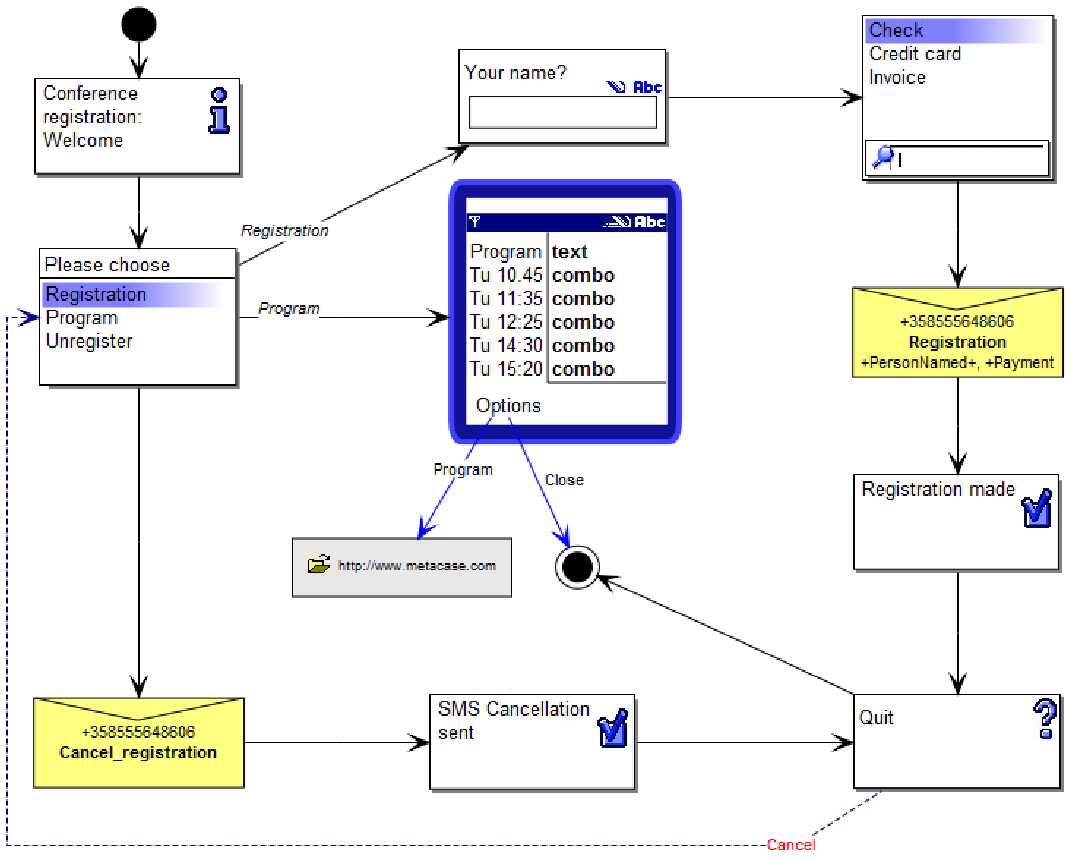


Figure 1-1. Conference registration application

As you can see from the model, all the implementation concepts are hidden (and are not even necessary to know at this stage). Developers can focus on finding the solution by using the phone domain concepts, such as Note, List, Form, Send SMS and Query.

The modeling language also covers phone domain rules, which prevent developers from making illegal designs. For example, in S60, it is typical that after sending an SMS message, only one UI element or phone service can be triggered. Accordingly, the language allows only one flow from the SMS element. This means that application developers do not need to master the details of the S60 architecture and programming model. If you understand the phone UI and services it can provide, you can start developing cellular phone applications.

Finally, a developer can run a generator to produce code and execute the application in an emulator. If you have not installed the SDK and Python Framework, you may still inspect the generated code. Figure 1-2 illustrates how the application is executed in an emulator. This application is generated from the model illustrated in Figure 1-1. By default the generator expects that the emulator is installed in the default directory of the SDK installer. You may modify the target directory for the Python code by using the Generator Editor provided by MetaEdit+.

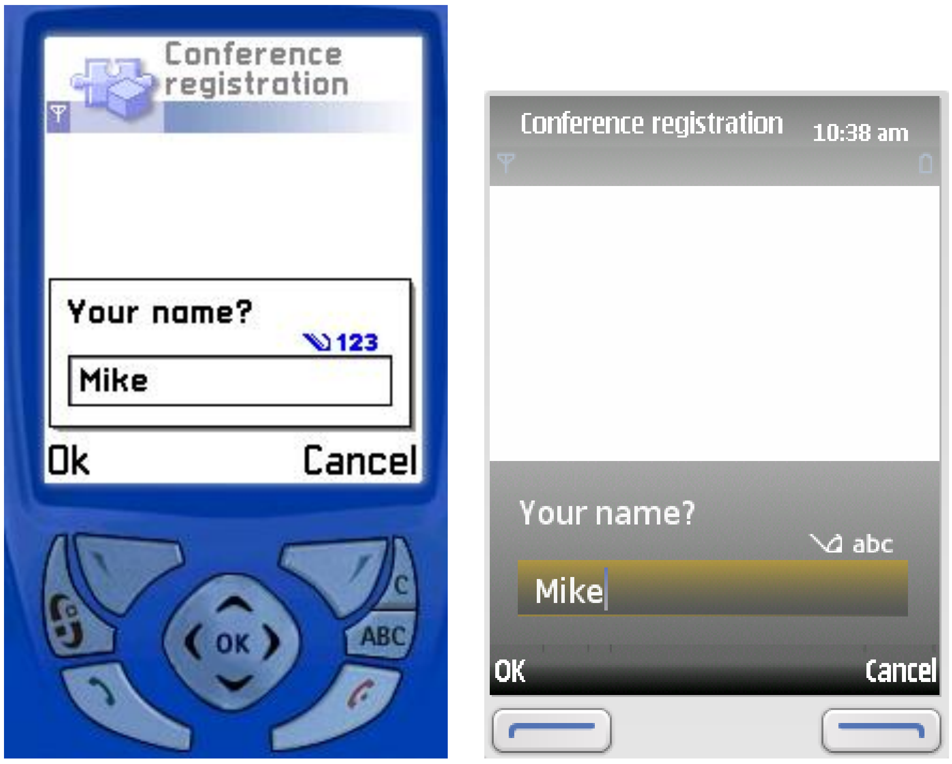


Figure 1-2. Running the generated code in an external PC emulator and in the phone

The generated code uses the services that the S60 platform and its framework provide. After design, there is no need to map the solution to implementation concepts in code or to modify the generated code. The generated applications should also have better quality, since typical errors found in manual programming do not occur anymore.

In addition to generators that produce code, a developer may also produce final documentation of the application or run design checking by using specific generators.

1.3 ABOUT THE S60 PHONE MODELING LANGUAGE

The modeling concepts of the language are shown in the editor's toolbar. They can also be seen from the Types menu. Figure 1-3 shows the contents of the Types menu and summarizes the modeling concepts with their notational symbols.

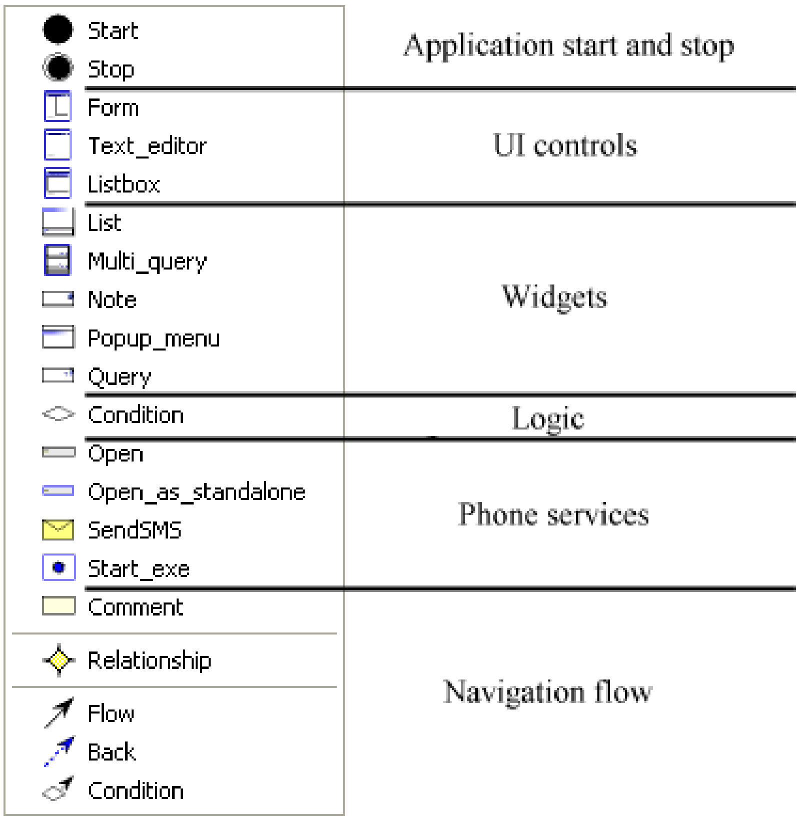


Figure 1-3. Modeling concepts for S60 phones

The modeling concepts are grouped as follows:

- ‘Start’ and ‘Stop’ concepts are used to specify application start and end states. Their semantics are close to the Start and End states as found in state machines.
- UI controls (Form, Text editor and Listbox) fill the whole display. They have their own state behavior and richer internal structure than other UI elements.
- Widgets represent the various kinds of dialogs that are available for the application.
- ‘Condition’ is used to specify logic and extra rules for the navigation flows.
- Phone services represent the concepts that access the Symbian and S60 phone services via the Python API. These include accessing files stored in the phone, browsing the web, sending SMS text messages, making calls and accessing other built-in phone applications, such as the calculator, calendar or camera.
- Navigation flow is specified by using three different kinds of relationships:
    - ‘Flow’, which describes the normal navigation flow through the application.
    - ‘Back’, which is used to specify navigation when canceling, overriding the default cancel policy.
    - ‘Condition’, which is used when application logic requires explicit conditions that could not be described using the higher-level modeling concepts.
- ‘Comment’ is used to attach free textual descriptions, visible in the design model.

2 Working with DSM for S60 phones

In this chapter, we discuss how to access the example in MetaEdit+ and how to work with it. First we play around with an existing application design, and then we modify the application using the modeling language. Finally, we generate the modified application and run it in a PC-based emulator.

2.1 ACCESSING THE S60 PHONE EXAMPLE

To access the S60 phone example, start MetaEdit+, choose the demo repository, select ‘Mobile UI’ from its project list, and login as usual. The S60 phone example can be accessed with the normal MetaEdit+ browsing and modeling tools like the Graph Browser and the Diagram Editor. Note that the models we are interested in here are under the “1. Symbian S60 Python: Project Model” graph. In addition to the Python-based example, there is also a second similar language for S60 generating C++, and a third generic mobile UI example that does not generate code.

2.2 PLAYING AROUND WITH THE S60 PHONE LANGUAGE

We start by inspecting the existing design models. They are listed in the Graph Browser in the main MetaEdit+ window. Double-click the graph named ‘Conference registration: Application’ to open it in a Diagram Editor. This is the application we looked at in Section 1.2.

The application logic is drawn as flow relationships from one start state, via the various UI actions and services, to stop states. You may inspect the properties of any model element by double-clicking it in the diagram, or directly view and edit the selected element’s properties in the Diagram Editor sidebar. You may also access operations related to each model item by first selecting the element and then opening its pop-up menu.

2.3 MODELING TO CHANGE THE APPLICATION

The next step in working with the modeling language is to change the current application logic by adding new elements to the model and changing the navigation flows. The change request for our modeling tasks is as follows:

a) If the user wants to stop the registration process, control must go back to the start menu to select a new required action.

b) If the user wishes to cancel the registration, the application has to ask for a confirmation prior to sending the SMS message.

c) Search functionality for choosing a payment method can be removed as unnecessary, since there only a few options.

Let’s next model the application accordingly and then generate code to execute the modified application to verify the changes. The next sections show how to make the changes to the model.

2.3.1 Adding a new relationship for the navigation flow

The default policy to navigate backwards on cancel is to go to the previous UI element, excluding elements that perform irreversible actions like sending text messages, making a phone call or taking a picture with the camera. In our current design model pressing cancel while choosing a payment method follows this default policy: it returns to the Query that asks the user to fill out a name. This cancel behavior is not explicitly shown in the model, as it is the default behavior and therefore does not need to be modeled. Now, however, we need to specify that the application returns back to the start menu when cancel is pressed, instead of asking again for the user’s name.

To override the default cancel behavior, we need to add a new navigation flow back to the start menu. Since we already know the flow type, named in the language as ‘Back’, the fastest way to create a relationship is to click the ‘Back’ relationship type on the toolbar, then click the elements that we wish to connect.

To add a breakpoint as a vertex of the relationship line you may click the drawing area in the places where you would like to add a breakpoint. One such option is illustrated in Figure 2-1. You may also drag the relationship later, or add breakpoints to the line by select the line and dragging a point on it to a new position. Ctrl-click a breakpoint’s handle to remove it.

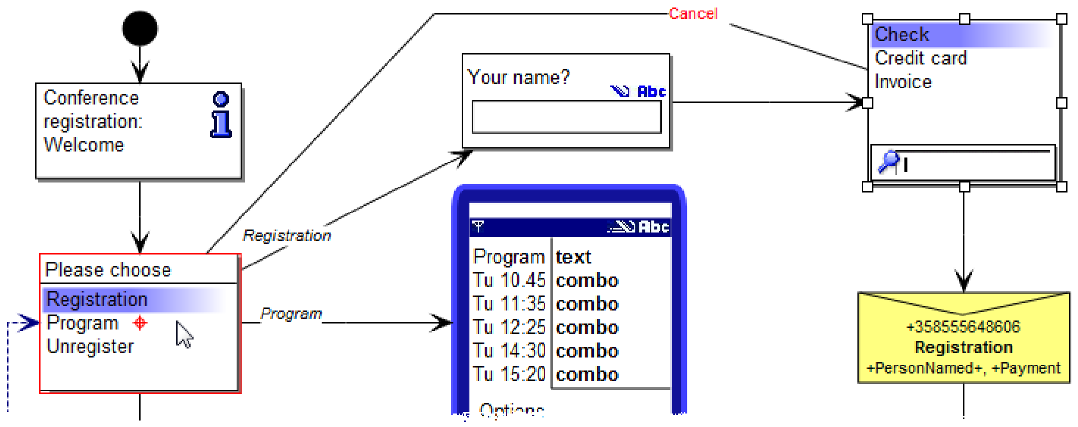


Figure 2-1. Adding a cancel navigation relationship to the application

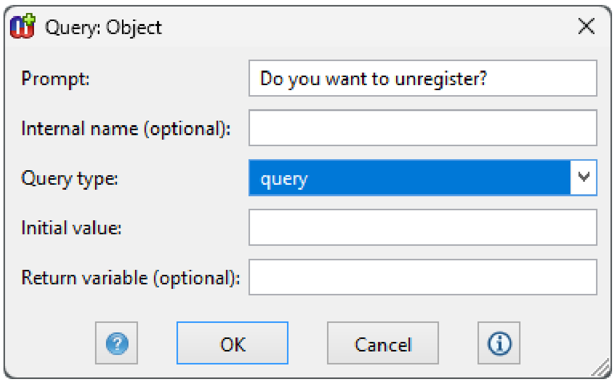
After connecting the elements a dialog is opened to specify the properties for the Back flow. In our case they are not needed so just press **All OK**. Now the design is updated and we could generate the code and test the application by pressing the ‘Build’ button in the Diagram Editor toolbar.

2.3.2 Adding a new widget element

The second request was to ask for a confirmation before sending a message for canceling the registration. Here we need to add a new dialog which asks the user to confirm. For this purpose, the ‘Query’ object is the most suitable. To add a query, click the ‘Query’ button on the toolbar or select it from the Types menu. Then click in the diagram into an empty place to add a new Query. This opens a dialog where we can specify the details of the element.

For a ’Query’ we can enter its ‘Prompt’ text (like ‘Do you want to unregister?’) and fill in other properties as needed. Here only the query type is important: Choose ‘query’ from the list to have a Boolean query. The Boolean query means that pressing OK in the application continues the flow and pressing Cancel goes one step back. The other property fields are not needed here: ‘Internal name’ allows to give a name for the query that will then be shown in the generated code (otherwise the generator automatically creates a name), ‘Initial value’ is not relevant as the selection is made directly by using the navigation buttons, and entering ‘Return variable’ stores the query answer in the named variable. In the conference application such information is not needed, unlike for example when sending the registration message which sends as arguments data stored in variables for payment method and registrant name.

After you have defined the two relevant properties, the dialog for specifying a query should look like Figure 2-2. Choose **OK** and close the dialog. This adds the created object to the diagram.


Query: Object

Prompt:	Do you want to unregister?
Internal name (optional):	
Query type:	query
Initial value:	
Return variable (optional):	

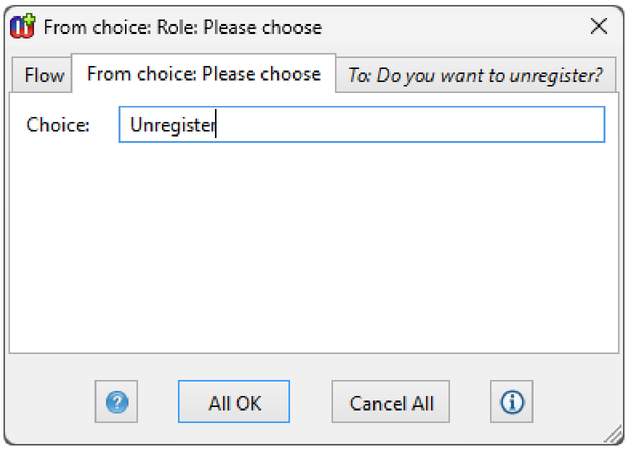
Figure 2-2. Adding a query to the application design

Next, you can delete the current relationship from the ‘Please choose’ Popup menu to the ‘Cancel_registration’ SMS message. Then move the created query between the Popup menu and the ‘Cancel Registration’ SMS sending object and connect it with them. Relationships can be created in many ways: let’s use a way which is not based on pre-selecting a relationship type. Select the ‘Please choose’ Popup menu object, open its pop-up menu and choose **Connect**.... Next click the query element you created to specify the target for the relationship. This opens a dialog asking for a relationship type. As we did not pre-select a relationship type from the toolbar MetaEdit+ asks to choose from those that are legal between these object types. Choose ‘Flow’ by double-clicking it, or selecting it and pressing **OK**.

This creates a relationship in the model. If the relationship or its roles have properties, a new dialog opens allowing their entry. In our case a Flow from a Popup menu needs to specify which menu item in the Popup menu takes this path, so fill in ‘Unregister’.

Click the tab ‘From choice: Please choose’, enter the choice value ‘Unregister’ as illustrated in Figure 2-3 below and press **All OK** in the dialog.

→ *Strictly speaking, the choice value is not mandatory here because the other possible choices are already specified on other Flows exiting the Popup menu. The code generator thus allows one path to leave its Choice empty, and that path will be followed if a chosen menu item is not found in the other paths. If more than one choice value is left unspecified, a model check will report a warning and point out data that needs to be specified.*


From choice: Role: Please choose
Flow
From choice: Please choose
To: Do you want to unregister?
All OK
Cancel All

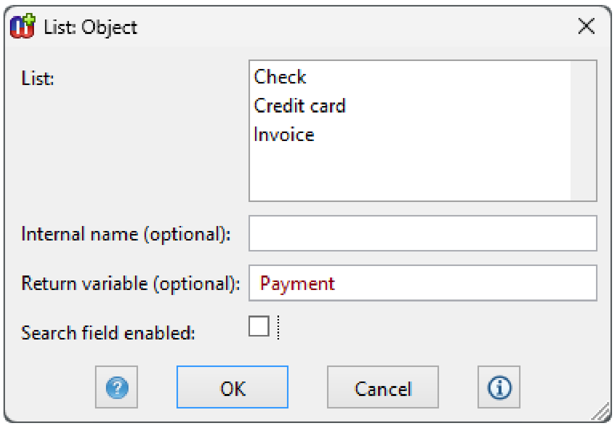
Choice: Unregister

Figure 2-3. Entering a choice value for the navigation path

Finalize the navigation flow by connecting your new query to the SMS element sending the 'Cancel_registration' message. You can create the relationship as we did earlier, or first click the 'Flow' relationship type from the toolbar and then click the elements in the navigation order: first the query as a source and then the SMS send as a target.

2.3.3 Changing a model element

Finally, we need to remove the unnecessary search field from the list element that provides the payment options: as there are just three payment options, the optional functionality for searching long lists using the phone keyboard is not needed here. To change the design model accordingly, you need to modify just one property of the List object. First double-click the List object for choosing the payment method and then deselect the 'Search field enabled' property (see Figure 2-4).



List: Check, Credit card, Invoice

Internal name (optional):

Return variable (optional): Payment

Search field enabled: ☐

Figure 2-4. Modifying properties of a design element

After the modifications the application design should look like in Figure 2-5.

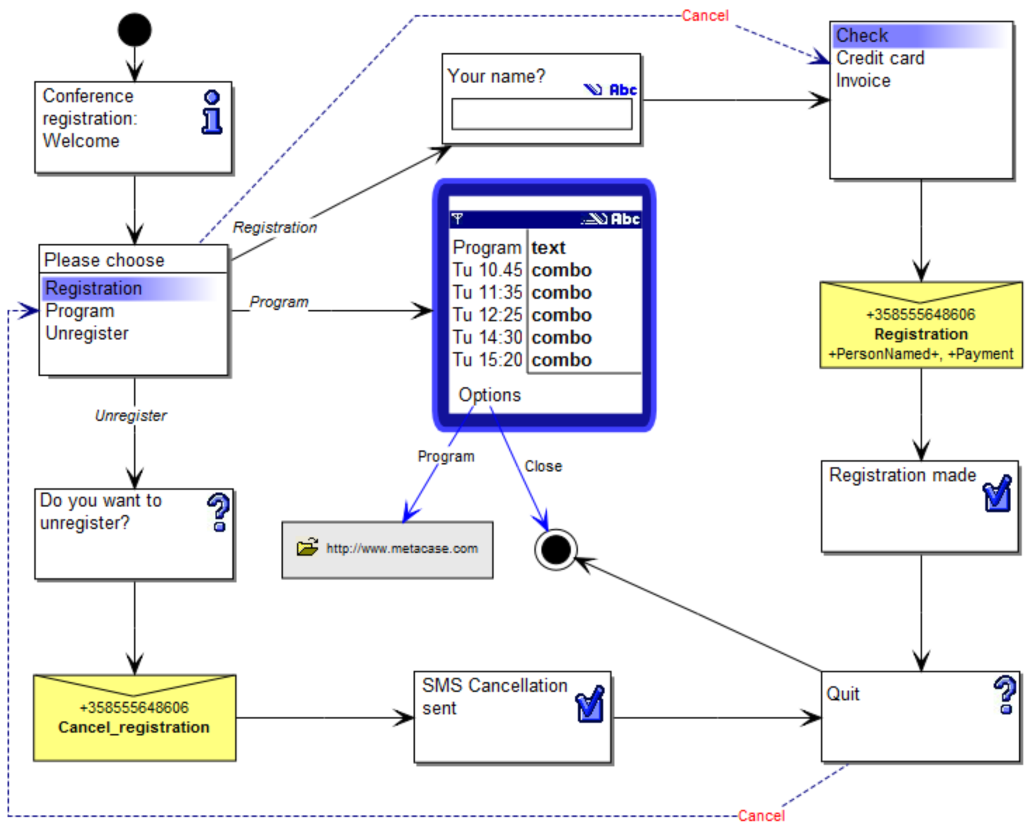


Figure 2-5. Modified conference registration application

2.3.4 Generating the application code and running it in an emulator

DSM is made to support agile development: At any stage we could have produced the code and tested the application by simply running the generator. After making all three changes, we can now generate code to run the application in a PC emulator. If the emulator is not installed (see instructions in the preface) then you may only inspect the generated code.

To execute the application simply press the ‘Build’ button in the Diagram Editor’s toolbar. This produces the code and starts the emulator so you can run the generated application. After the PC emulator starts, choose Python from the phone desktop (you may need to scroll down to see the Python icon). Then press the **Options** key and choose **Run script** from the menu. This opens a list of available applications. Choose ‘Conference registration’ and press **OK**. You may now use the application to test the changes you made. Figure 2-6 illustrates the query we added.

→ *In the emulator, SMS sending does not work as it does not have access to a cellular network. The generated applications instead show a note that displays the content of the SMS message. For deployment in the product, you can modify the SMS sending part of the generator and remove the note code and comments from the actual SMS sending message. See the code generator definition for details.*

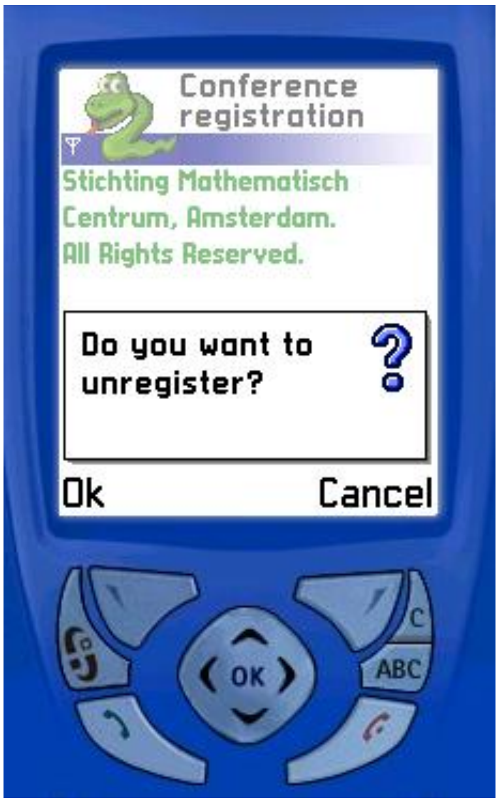

Conference registration
Stichting Mathematisch Centrum, Amsterdam. All Rights Reserved.
Do you want to unregister?
Ok
Cancel
OK
C
ABC

Figure 2-6. Running the modified application after code generation

If the emulator does not start, it is most likely because the Nokia S60 SDK emulator is not installed in the default location or the newer version of the SDK uses different installation paths. You may copy the generated code into the Python directory in the SDK or, as a better way, modify the generator paths in the Generator Editor. To do this, choose **Graph | Edit Generators…** in the Diagram Editor. In the Generator Editor, choose the generator named 'Autobuild' from the list on the left. You will find it in the tree structure under '!Build'. Its contents looks like this:

```
$dir='C:\Symbian\8.0a\S60_2nd_FP2\'
subreport '_Generate Python file' run
external $dir 'epoc32\release\wins\udeb\Epoc.exe' execute
```

Change the two paths according to your installation. The first path specifies the location of the Python scripts and the second the location of the emulator. For example, if you have installed the Symbian 5th Edition SDK the paths should be as follows:

```
$dir='C:\S60\devices\S60_5th_Edition_SDK_v1.0\'
subreport '_Generate Python file' run
external $dir 'epoc32\release\winscw\udeb\epoc.exe' execute
```

Similarly, the location of the generated Python .py files may need to be changed. In the generator called '_Generate Python file' change the directory accordingly. For example, in the Symbian 5th Edition SDK the directory should be:

```
$dir 'epoc32\winscw\c\data\python\' id '.py' write
```

You may also run other generators, like those that produce documentation, run metrics and model checking. You are also welcome to make other changes to the application, inspect other available design models or to create totally new applications – from model to executable code. We leave this part to you and move next to look at the implementation of the DSM solution.

3 Creating the DSM solution in MetaEdit+

We have now used the S60 phone language. Next, we shift to the language and code generation definition.

3.1 Language Definition

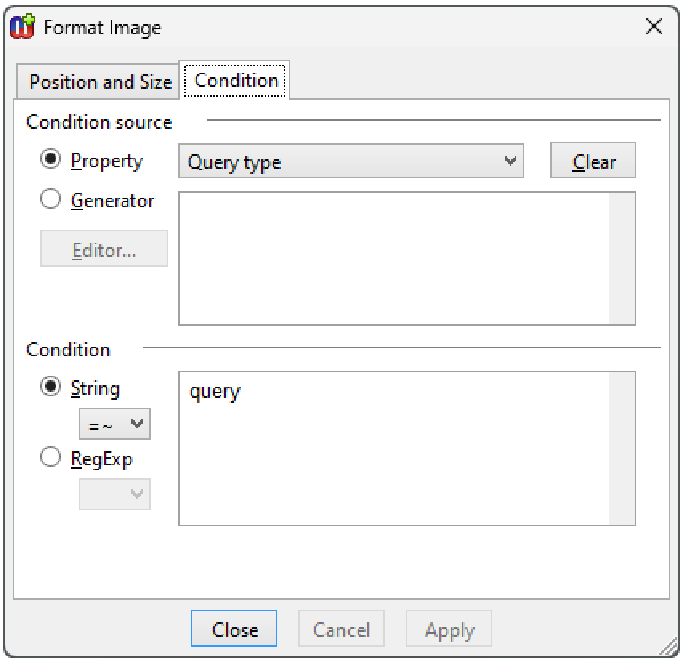
The MetaEdit+ Workbench User's Guide describes in detail the functionalities for defining modeling languages. These metamodeling functionalities are also discussed in tutorials, such as the Evaluation Tutorial, the Watch Example and the Graphical Metamodeling Example. We focus here on two particular aspects of language definition: importing external graphics for the language's notation and using ports.

3.1.1 Importing external graphics into language symbols

The DSM language for S60 phones is based on a 1:1 visualization of the actual phone widgets. The graphical elements used in the phone UI can thus be applied directly in the modeling language. MetaEdit+ allows external graphical elements to be imported either as bitmaps (BMP, GIF, JPG, PNG) or vector graphics (SVG). Next, we show one scenario of how to import graphics as symbols to represent a modeling concept and give conditions for its visualization.

In S60, the different kinds of query dialogs are distinguished by a dedicated icon for each, e.g. a question mark for a Boolean query. This symbol, available as a bitmap, can be imported into the Symbol Editor by choosing **Symbol | Import Bitmap…** and then selecting the desired file.

Once the symbol element is added, its representation can be made conditional based on the values given by a modeler. All the icons will thus be on top of each other in the symbol, but in any given Query object only one will be shown. In the case of the question mark it should only be shown when the modeler selects 'query' as the value for the 'query type' property. To add such a constraint, select the question mark symbol in the Symbol Editor and open its formatting settings by choosing **Format…** from the pop-up menu. Then go to the condition tab (see Figure 3-1) to specify the condition.



Condition source

(● Property) Query type

( ) Generator

Condition

(● String) =~

( ) RegExp

query

Figure 3-1. Specifying symbol condition

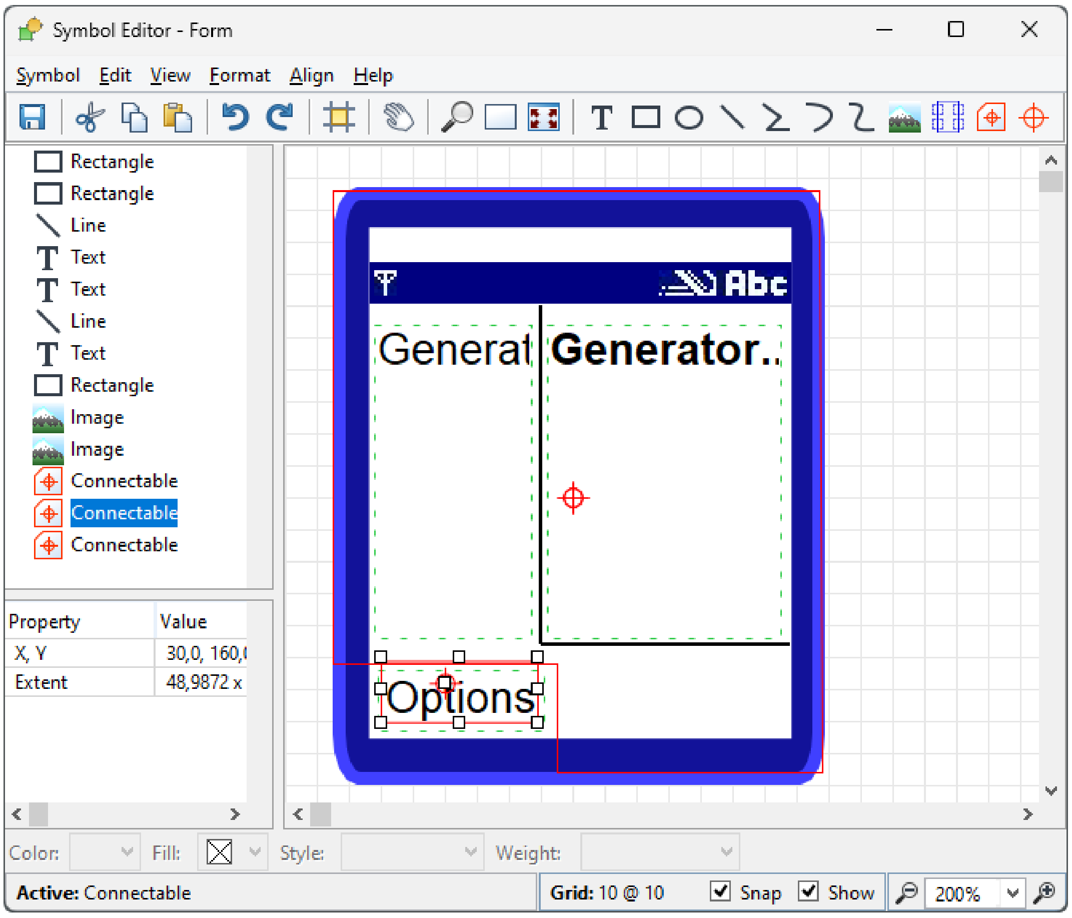
Symbol conditions are specified by first selecting where to look in the model, and then what should be found there for the condition to be fulfilled. The condition source can be based on a property or on the output of a generator. A generator-based condition source allows you to specify almost any condition for symbol visualization. Often, like here, the condition is based on one property only.

As shown in Figure 3-1, a property is chosen to be the source and from the list the ‘Query type’ property is selected. In the condition part, the property value to expect is filled in as ‘query’. This is one of the values specified in the Query type list property in the metamodel. As a result, when a query type is chosen to be a Boolean query in the model, the question mark icon is shown. Similar conditions on the other icon symbol elements in Query are not met, are hence those icons are not shown.

3.1.2 Defining static ports

The S60 language uses static ports as representational elements so that main UI controls like Form can have relationships connecting to two distinct areas, with different semantics. UI controls have their own menus and their specification in the language is implemented using a port. When creating a relationship from a form in the model you can move the mouse over the symbol and see that it has two connectables: a larger one for the main symbol and a smaller one for the Options menu.

To see how the representational port is defined, open the Symbol Editor for the Form object (see Figure 3-2). Then choose the connectable around the Options menu by selecting its target point crosshair. Selection handles should be shown around the edge of the connectable as usual, plus an extra one on the target point crosshair. If the crosshair shows no selection handle, you may have chosen the Options text element by mistake. The Symbol Editor shows the selected element in its sidebar and in the Active field of the status bar.

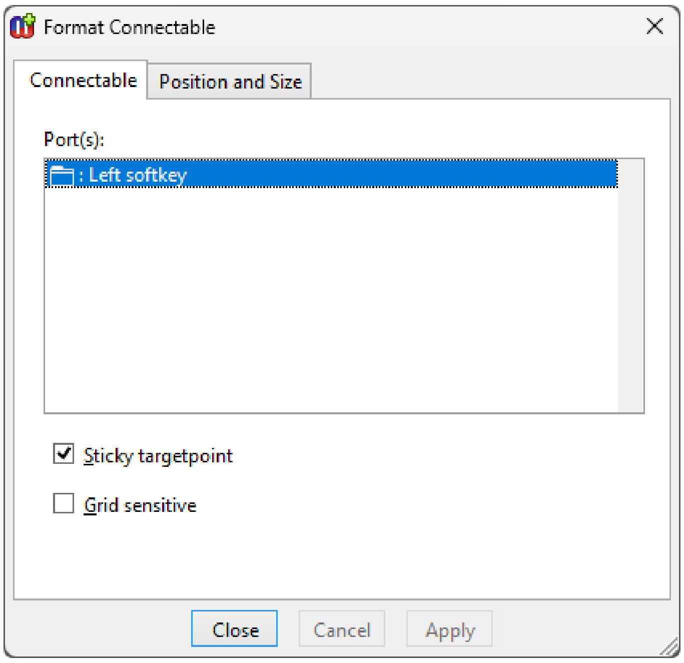


Property	Value
X, Y	30,0, 160,(
Extent	48,9872 x

Active: Connectable

Figure 3-2. Form symbol

Open the **Format…** menu from the pop-up menu to see the port definition given for the selected connectable. Figure 3-3 shows that the connectable uses a port of the 'Left softkey' port type: pressing the left soft key in the real S60 phone opens the Options menu.



Port(s):
: Left softkey

☑ Sticky targetpoint
☐ Grid sensitive

Figure 3-3. Relating ports to connectable

Port types are defined in the Port Tool, similarly to other language concepts. The instances of port types are created in this dialog with **Add…**; a given port type can have several instances, and a given instance can be used in several connectables, each in different symbols. In this case, there is only one instance, and the port is only used only for representational purposes not language rules, so the 'Left softkey' port does not have any properties. For more details on Ports see MetaEdit+ Workbench User's Guide.

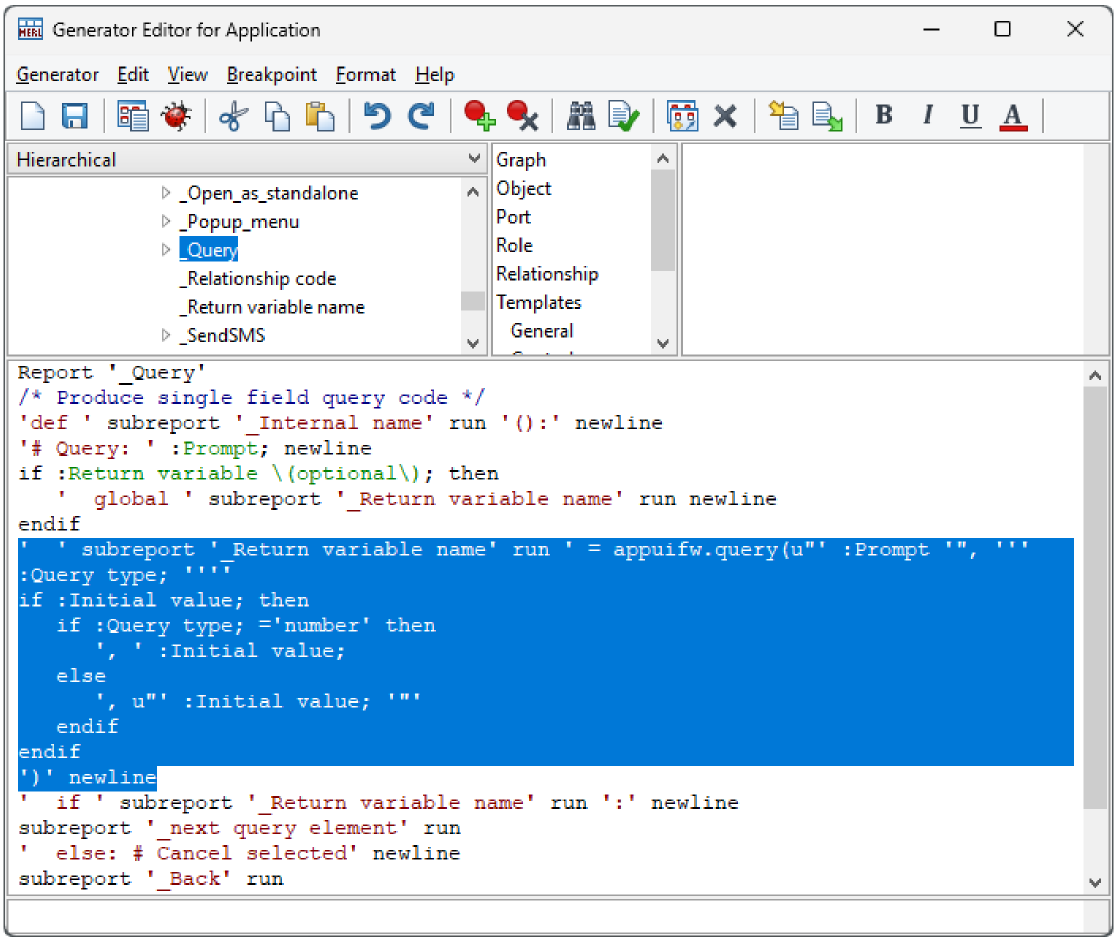
3.2 CODE GENERATION: RELATING MODELS AND CODE

Often it is not possible to generate all code directly from models. We may have existing code and 3rd party libraries, or not be able to make the DSM solution specify all the functionality using domain concepts. In this section we show some approaches for integrating models with manually written code, using the S60 language as an example.

3.2.1 Generator provides integration with the component library

Usually, the best approach is to create a component library or a component framework that provides prefabricated software building blocks that can be configured and used when developing applications. This was exactly the case for S60 as its services are made available via the Python framework. The generated code then calls the services provided by this framework, rather than having to generate much longer lower-level code. Creation of such a component library usually does not necessarily require a lot of effort, as such components may already exist from earlier development efforts and products, and just need a little tweaking or wrapping.

To illustrate how a code generator calls the services of the framework, let's highlight the case of Query. You can see the code generator for the Query object by selecting **Graph | Edit Generators…** in the Diagram Editor. This opens the Generator Editor. Choose the generator named '_Query' from the list on the left. You will find it in the tree structure under '!Build'. You can switch the Hierarchical view to the Alphabetical view in the pull-down menu above the list, find _Query and then switch back to see it in the hierarchy. Your view in the Generator Editor should then look like Figure 3-4. It shows the complete generator definition for producing Query code.



```
Report '_Query'
/* Produce single field query code */
'def ' subreport '_Internal name' run '():' newline
'# Query: ' :Prompt; newline
if :Return variable \(optional\); then
   '  global ' subreport '_Return variable name' run newline
endif
'   ' subreport '_Return variable name' run ' = appuifw.query(u"' :Prompt '", '''
:Query type; ''''
if :Initial value; then
   if :Query type; ='number' then
      ', ' :Initial value;
   else
      ', u"' :Initial value; '"'
   endif
endif
')' newline
'  if ' subreport '_Return variable name' run ':' newline
subreport '_next query element' run
'  else: # Cancel selected' newline
subreport '_Back' run
```

Figure 3-4. Calling a service of the underlying framework

The highlighted area in the figure shows the part of the generator that makes the actual call to the Python framework. In other words, it takes the data from the model to produce the call to the API. The code produced by the highlighted generation definition looks like this:

```
PersonNamed = appuifw.query(u"Your name?", 'text')
```

This code is actually taken from the Conference registration application where a query element is used for requesting the user's name (see Figure 1-1). At the start of the highlighted part of the generator, the '_Return variable name' subreport picks up the variable name from the model, in this case 'PersonNamed', or if none is specified it automatically creates a new variable. The rest of the first highlighted line generates the call to the framework. First is the fixed text reference to the appuifw module and its query service: a Query will always generate that call. Next the generator fills in the call's parameters, fetching them from the query object in the model: its Prompt property, 'Your name?', and its Query type property, 'text'. The rest of the highlighted part of the generator includes the initial value to the call if one is specified in the model.

As this small example illustrates, in the DSM solution the metamodeler uses his knowledge of Python, the phone framework and the programming model, to raises the level of abstraction and hide this complexity for the modelers. The modelers can thus concentrate on the applications they are making, using the higher-level concepts of the modeling language.

3.2.2 Referring to libraries from the models

The S60 language also allows Python code to be added directly in the models when needed. The places where manual code can be added are restricted to just those where it makes sense. As well as writing code manually directly in those places in the models, modelers can also use the generally more efficient approach of adding a manual reference to library code.

Consider the calculation example shown in Figure 3-5, the Calculate model in the Mobile UI project. This small application asks for two numbers and shows their sum in a note dialog. The calculation and result display is not specified by using the modeling concepts but rather entered as manual Python code in the note. The generator simply outputs the manual code at the appropriate place in the generated code.

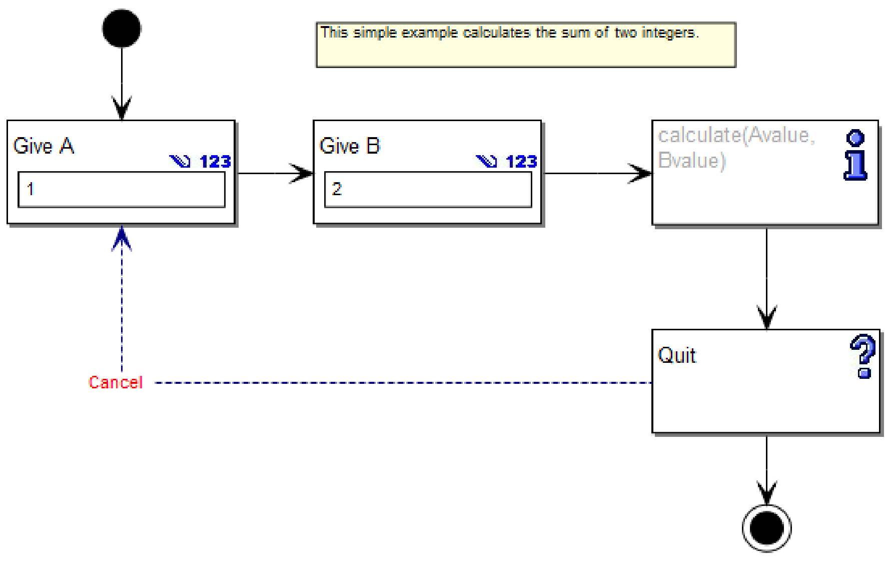
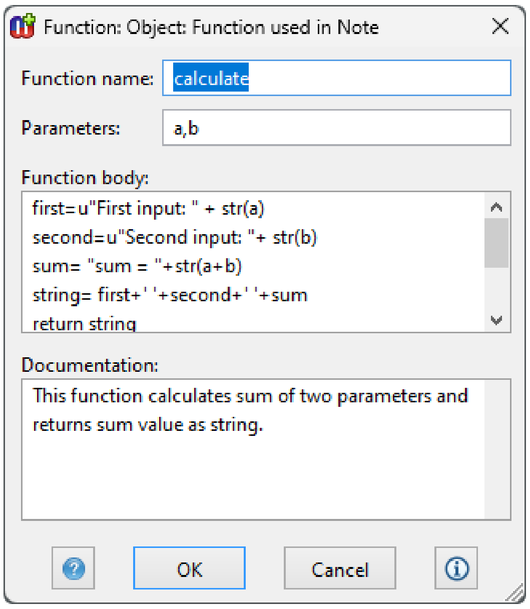

This simple example calculates the sum of two integers.
Give A
123
1
Give B
123
2
calculate(Avalue, Bvalue)
Quit
Cancel

Figure 3-5. A design model referring to a code library

Double-click the note element to see its properties. Rather than showing text inside the Note element its content is now treated as a code. We could simply write the calculation and display code inline in the ‘Text or code’ property, but here we choose to show how such code can be made more reusable by putting it in a function library. The code we want is saved into a function, which can be used in multiple model elements, and here we simply specify the arguments to the function. In this case, we have called our function ‘calculate’, and we provide it with two arguments, Avalue and Bvalue. These have been defined as the return variables in the two queries made before the calculation, and match the two parameters a and b defined for the ‘calculate’ function in the library.

Libraries are usually considered to be either black-box or white-box. In the case of a black-box library we see just the interface (here: ‘calculate (a,b)’) but not its implementation. For the S60 case the language is defined to be white-box: the contents of these library function can be seen and modified by the modeler. In the property dialog or sidebar property sheet for the Note, double-click the Function used property ‘calculate (a,b)’ to see its implementation. Figure 3-6 shows the implementation and description of the calculate function.



Function name: calculate

Parameters: a,b

Function body:

```
first=u"First input: " + str(a)
second=u"Second input: "+ str(b)
sum= "sum = "+str(a+b)
string= first+' '+second+' '+sum
return string
```

Documentation:

This function calculates sum of two parameters and returns sum value as string.

Figure 3-6. Selecting a library function

The S60 language allows us to define functions while modeling, and also to import them from an existing library into MetaEdit+ by using the API functionality. We could choose to have such functions imported completely, or to make only their public interface available in models. In the case of the S60 language, the code generator expects that Functions are imported completely, so their content is also generated into the application code. Alternatively, a code generator could import or include the function from the library during generation.

3.2.3 Entering code directly in models

Instead of referring to or importing libraries, it is always possible to have one or more language elements just for writing code directly into the models. This is obviously not as good an option as having a framework or libraries, but in selected places it can be useful. In the case of the S60 language, a Form has for instance an optional ‘save validation’ function that is run to check that the entries made in a Form are correct. In the modeling language this ‘save validation’ function is entered in a textual property.

For a more complex case let’s look at the condition element in the S60 language. Condition allows us to specify additional rules, checks and other navigation information that goes beyond the direct DSM support for conditionality implicit in the navigation structure. In the ‘Restaurant finder’ application (see Figure 3-7) a condition is used to check that the given zip code is legal. In our case, it checks that a zip code has 5 digits.

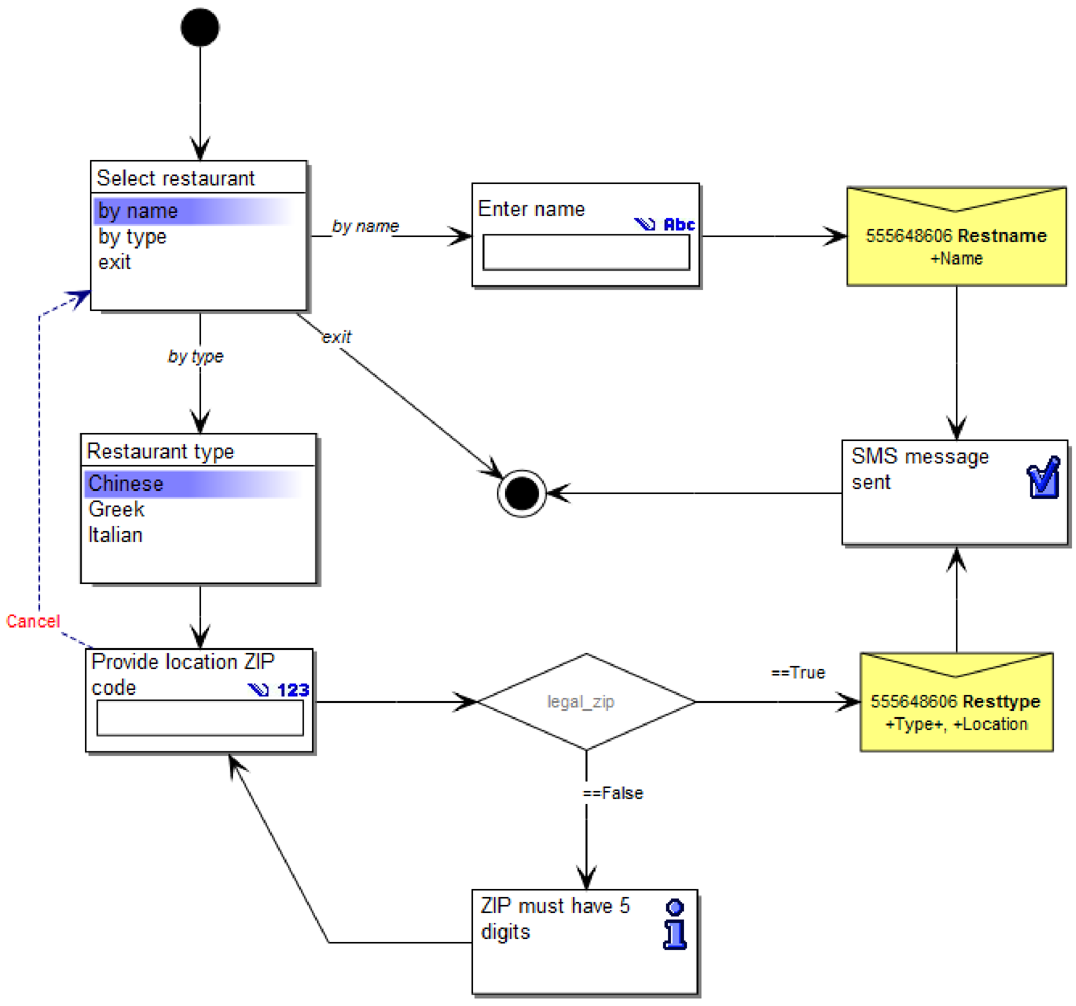
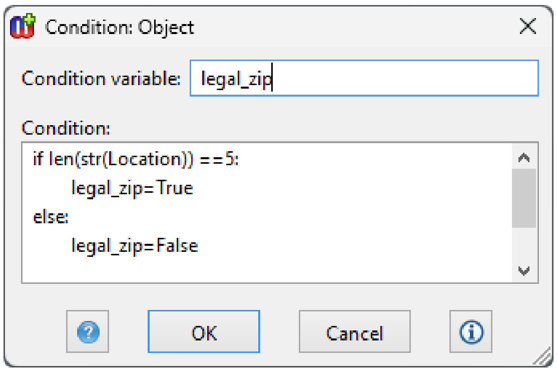


Figure 3-7. Restaurant finder application with a condition for ZIP code validation

The checking behavior is implemented in the diagram in a condition named 'legal_zip'. Double-click it to see its details as shown in Figure 3-8. The 'Condition' object has two properties: a 'Condition variable' and 'Condition'. The code is entered as a Python script so that it can be used directly by the code generator. The code returns either True or False. This value is then used for guiding the navigation: if the length of the zip code is 5 characters a SMS message is sent, otherwise a note is opened informing that 'ZIP must have 5 digits', and navigation returns to the ZIP code entry.



Condition variable: legal_zip

Condition:

```
if len(str(Location)) ==5:
        legal_zip=True
else:
        legal_zip=False
```

Figure 3-8. Restaurant finder application with a condition for ZIP code validation

Instead of having code written directly, we could also have made the language allow library functions similar to the calculate function that we discussed earlier. This could build up a library of common conditions that would then be reused in design models.

The text for these manually written code properties can be edited directly in the property widget in the property dialog, which suffices for short, simple pieces of code. The text property can also be opened in its own text editor window, either by double-clicking the property in the sidebar or choosing **Editor** from the property field in a property dialog. You can also change the Options settings in the MetaEdit+ main window to specify the use of your preferred external text editor: see the MetaEdit+ User's Guide for details. For any substantial code, it is of course better to edit it in a dedicated IDE, and simply refer to that function by name in the model.

3.2.4 Regeneration with protected blocks

The final option to integrate manual code and generated code is to use protected blocks. These allow you to generate partial or default code which can later be modified manually, without those manual changes being lost when the code is regenerated. To demonstrate this approach we will alter our current generator to add a protected block for Conditions.

→ *Note though that the S60 language did not take this approach: protected blocks mean that the final code is always a combination of two sources, the model and the manually edited code file. While editing generated code is often the first solution that comes to mind to developers used to writing all code manually, in practice the extra effort of maintaining two sources in version control tends to outweigh the benefits, particularly when compared to other ways of integrating manual code and generated code.*

While defining protected blocks, we need to consider two things: to identify the parts of the code where manual changes must be allowed (Condition in our case), and to define how those blocks are delineated in the code. The basic approach is to make all protected blocks in the file follow a consistent format, with a header and footer starting with a string not found elsewhere in the file, so MetaEdit+ can reliably recognize the protected blocks. Each block's header will contain an ID unique in the file, so it can be matched up when regenerating. Each block's footer will contain the MD5 checksum of the generated content of the block, so MetaEdit+ can recognize if the content has been edited.

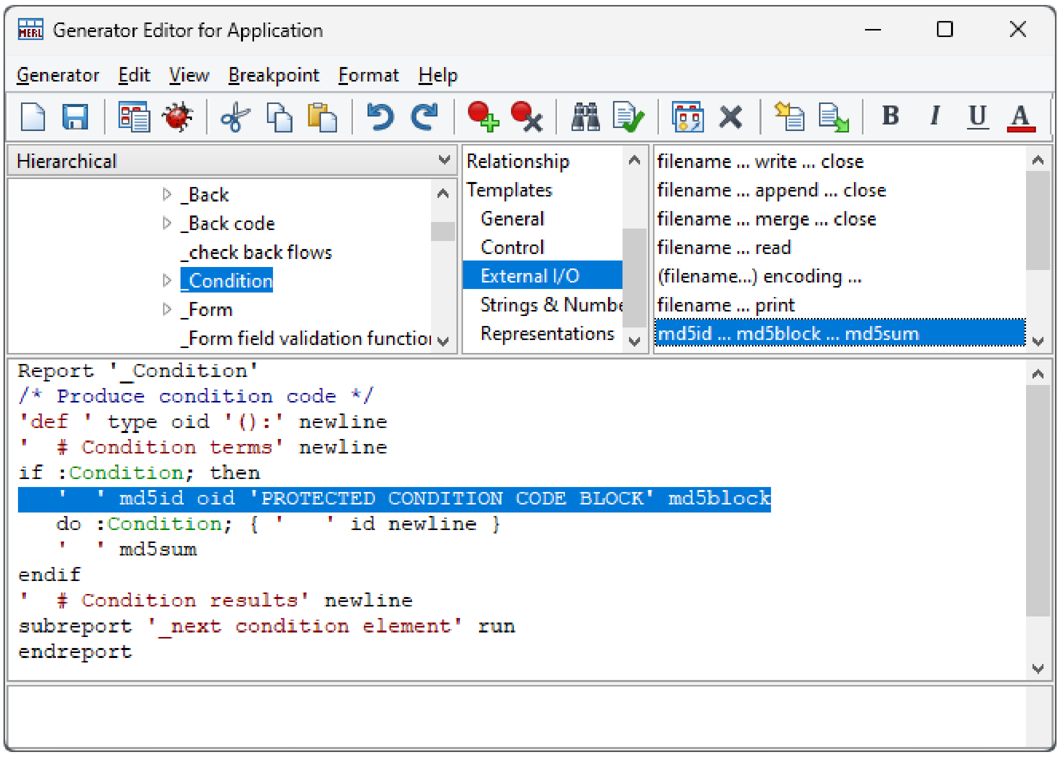
Protected blocks must be defined so that their header and footer are legal syntax in the target language, generally by making them comments. MetaEdit+ provides a default prefix and postfix, `/* MEPMD5 … */`, but for Python we can override those comment delimiters. Consider the following generator definition for setting up generation to a Python file with protected blocks:

```
filename subreport '_default directory' run id '.py'
     md5start '# MEPMD5 ' md5stop newline merge
     subreport 'Generate Python script' run
close
endreport
```

The first line of the generator is unchanged and specifies that we are generating to a file named using the model `id` and the '.py' extension for a Python file. The second line defines the prefix and postfix for protected block labels and checksums. These will be used throughout this `filename…merge…close` command. Here the only change to the default values is to use just '`#`' at the start to delimit a comment in Python. Note the normal `write` command has been replaced with `merge`: this tells MetaEdit+ to merge the protected blocks in the existing file on disk, rather than ignore its current contents and overwrite it.

Next, we need to modify the generator which produces the code to be written as a protected block. For the condition code we must open the Generator Editor and its generator named '_Condition'. In this generator, we enclose the generation of the protected block in an `md5id`...`md5block`...`md5sum` command, with the existing generator code in the latter part.

Figure 3-9 shows the generator definition when the protected block definition is added into the generator. The highlighted line starts the block with the `md5id` keyword, followed by the unique id within the file. Here, the `oid` refers to the unique identifier MetaEdit+ gives for all model elements. The line also includes comment text so that the block can be identified from the generated code. The `md5block` keyword finishes the identifier and starts the section for the actual protected block contents. The '`do`' clause then takes the value from the model and indents every line to follow Python conventions to indent code inside a function. Finally, `md5sum` is used to end the protected block, and will output the MD5 sum in the comment syntax specified earlier.


Generator Editor for Application
Generator Edit View Breakpoint Format Help
Hierarchical
_Back
_Back code
_check back flows
_Condition
_Form
_Form field validation functio
Relationship
Templates
General
Control
External I/O
Strings & Numbe
Representations
filename ... write ... close
filename ... append ... close
filename ... merge ... close
filename ... read
(filename...) encoding ...
filename ... print
md5id ... md5block ... md5sum

```
Report '_Condition'
/* Produce condition code */
'def ' type oid '():' newline
'   # Condition terms' newline
if :Condition; then
   '    ' md5id oid 'PROTECTED CONDITION CODE BLOCK' md5block
   do :Condition; { '    ' id newline }
   '    ' md5sum
endif
'   # Condition results' newline
subreport '_next condition element' run
endreport
```

Figure 3-9. Generating code with protected blocks

With these updates, the generator outputs the Condition with a unique header and an MD5 checksum footer. If changes are made into the block, its content is not generated. In case of the Restaurant finder application the generated code for the condition looks like:

```
def Condition25_1763():
     # Condition terms
     # MEPMD5 25_1763 PROTECTED CONDITION CODE BLOCK
     if len(str(Location)) ==5:
          legal_zip=True
     else:
          legal_zip=False
     # MEPMD5 4a958480fd0245a0626fd58c2be1f277
```

→ *See MetaEdit+ Workbench User's Guide for details on using regeneration support.*

4 Conclusion

In this example, we have demonstrated a DSM solution for phone application development. With the domain-specific language we can model applications using phone concepts and execute them in a PC emulator or in the real phone device.

On the DSM language definition side we focused on a few areas of language design: ports and importing external graphics to language notation. On the code generation side we described some options for integrating manual code with the code generated from the models.

The modeling language is implemented as any other modeling language in MetaEdit+. It is completely open and thus it can be freely extended to cover additional requirements of modeling or code generation. You could for example modify the generator to produce Symbian Java code from the same models, or extend the modeling language to cover a larger part of the phone framework than that provided by Python for S60. The choice is yours because with DSM you control both the language and the generators.