Dipartimento di Informatica
Università degli Studi di Verona

Rapporto di ricerca
Research report
July 2012

RR 88/2012

Multi-Platform Design of Smartphone Applications

Giulio Botturi
Davide Quaglia

Questo rapporto è disponibile su Web all'indirizzo:
This report is available on the web at the address:
http://www.di.univr.it/report

In 1969 a 2 Mhz processor (Apollo Guidance Computer) took us to the moon. Nowadays, a 1Ghz processor does not seem enough to send an e-mail. In computer science, and in everyday life, low resources require smart solutions!

Contents

List of Figures

Introduction

Technological advancement in the miniaturization of integrated circuits has made possible the realization of small and smart devices, which we can find in everyday objects. These devices provide computational capabilities comparable to those that only a decade ago were offered by the high-level personal computers, and great progresses have been made in terms of size and energy consumption. An example of a device that has benefited from these progresses was definitely the mobile phone. Since the introduction of the Global System for Mobile Communications (GSM), mobile phones have needed more digital computing power, to perform the task for which they were born. The evolution of this device, favored by its widespread use, has transformed it into a tool that integrates advanced features: from the ability to send text messages up to connect to the Internet. The availability of sufficient computing power and data storage, has made it possible to extend these capabilities to the installation of small operating systems, typically closed and proprietary. Consequently, the development of software has become an important aspect of these "smart" phones.
After a more detailed description of smartphones and their potentiality in Chapter 1, the Chapter 2 deals with the development of mobile applications, explaining the common approach proposed by the major producers of development environments. It also focuses on the portability problem of the application code because there are several different smartphones, each of whih has different embedded software. The Chapter 3 introduces the Model-Driven Design, a software development process driven by the activity of modeling the software system at two main levels of abstraction: platform independent and platform specific. The key idea is the creation of a model of the application, that is independent from a specific software platform. The Platform Independent Model is designed to capture all the structural and behavioral aspects of application, and will be introduced in Chapter 5. The chapter goes on explaining the proposed methodology to transform the independent model in several models, that are specific to a particular platform. The last step of this process is the generation of the application code for

each specific target platform. In this way it is possible to realize the basic structure of an application, starting by its abstract description, common to each smartphone platform. The Chapter 6 presents the experimental validation. First, a sample application has been modeled at platform-independent level. Subsequently, the transformation and code-generation rules have been applied. Finally, the obtained code has been compared with the same application, previously developed for Android and Windows Phone. In the end, the Chapter 7 reports considerations on the obtained results and the main difficulties encountered.

Chapter 1

Smartphones

Smartphones are high-end mobile phones that the user must be able to handle easily with one hand. At the same time, they must have a display large enough to reproduce rich Graphical User Interfaces. Operating System provides advanced Application Programming Interfaces (APIs) that allow running third-party applications with high integration with hardware and system functionalities. Typically, smartphones include high-resolution touchscreens, web browsers that can access and properly display standard web pages rather than just mobile-optimized sites, and high-speed data access via Wi-Fi and mobile broadband. Furthermore, device internal storage capacity can often be expanded by a memory card slot, that allow users to store a big quantity of application data (for example multimedia file, such as music or photo taken by the integrated camera). These devices must be pocket-size, and therefore sufficiently slim and small. At the same time, they must integrate a battery large enough to feed the integrated cpu, the display and all the other embedded functionality.
Today, all the major producers of mobile phone devices also offer smartphones, differentiated by quality, features and price. The offer is therefore very wide, and each manufacturer chooses which platform (or more than one) to adopt on their devices. Figure 1.1 shows the market shares of the different smartphone software platforms, often developed by the same devices manufacturers, on the market today.
Despite this variety of platforms, it is possible to recognize many key concepts about smartphones and smartphone applications. Next sections collects all this concepts in order to identify terminology and elements to talk about smartphone applications independently from a specific platform.

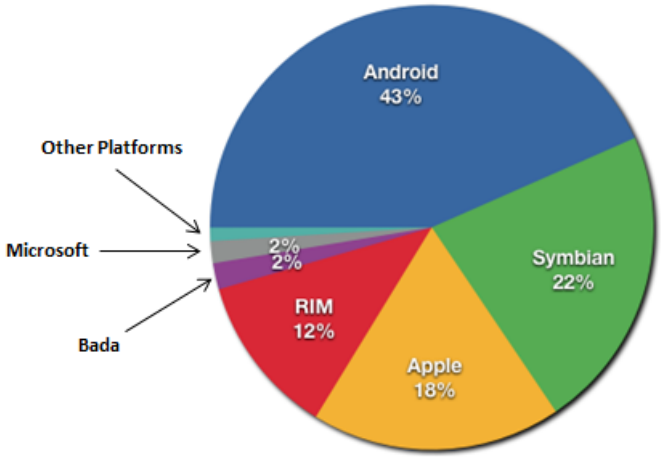


Figure 1.1: Worldwide Smartphone Sales to End Users by Operating System in 2Q11, according to *Gartner* [1]

1.1 Applications

The opportunity to develop user applications is a key feature for each smartphone platform and it is the main difference between smartphones and feature phones. The latters, indeed, typically come with a proprietary Operating System not ready to the installation of third-party applications.
The so-called "apps" are almost always programmed in an Object Oriented programming language, like Java, C-Sharp or ObjectiveC, and they are characterized by some basic elements.

- A list of permissions to access device features (integrated camera, external storage) and system functionality (network stack). During the installation phase the system informs the user what resources the application needs to access to.
- Application code
- A description (often in eXtensible Markup Language[1]) of the GUI layout for each screen.
- A table of resources externalized from application code, like images and strings. Externalizing resources brings many benefits. Indeed, it allows to provides alternative resources that support specific device configurations such as different languages or screen sizes, which becomes

[1]http://www.w3.org/XML/

increasingly important as more devices become available with different configurations.

Finally, when an application is ready to be distributed, developers can release them in online *App Store*, often managed by smartphone OS vendors themselves. Typically, the application may be released only after a registration fee and, in some cases, after an assessment of quality.

1.2 Graphical User Interface and User Interaction

Usually a smartphone application involves a number of screens, composed by a set of widgets, with which users can interact in order to perform several tasks.

Screens are the way the application presents its own features. One of them is specified as the "main" Screen, which is presented to the user when it launches the application for the first time. Each screen can then start another screen in order to perform different actions.

Widgets are the basic elements of a graphical user interface, like user controls in traditional desktop computer GUI. They are very standardized, and an experienced user on a particular platform can easily switch on a concurrent platform finding the same graphical components. The look and feel can be different and customized, but widgets are the same as well as the way to interact with them.

The simplest form of interaction is the touch of the screen with one finger. Multi-touch gesture is a very good example of advanced interaction, and refers to a feature employed by almost all touchscreen devices to perform various actions:

- A one-finger swipe is used to move one object between two points
- A pinch refers to pinching together the thumb and finger, and is used to zoom out on an image.
- and many others

A user who tries any smartphone expects to interact in this way, because it is very intuitive and commonly used.

Chapter 2

Mobile Application Development

Mobile application development is the process by which application software is developed for small handheld devices such as mobile phones, smartphones or tablet computers. Such devices are based on a mobile operating system that manages and controls the device resources, like in a classical pc architecture, with some differences due to limitations in power consumption, memory usage and computational power. Despite these differences, the development of mobile user applications involves the use of tools and development environments typical of the programming on personal computers.

2.1 Development Environment

The typical development environment includes the compiler, the debugger and several tools to manage large project and design graphical user interfaces more efficiently. When the target architecture is different from that on which we develop, there is the necessity of an emulator. Software Development Kit (SDK) comes with a target device software emulator, the essential tool that allows us to run and test applications as if we were using the physical device. In fact, emulator also reproduces the physical buttons, device features and functionality, such as network stack or external storage, and can simulate incoming calls.

2.2 Application Design

The software development process as we know it today is often driven by low-level design and coding. The typical process, as illustrated in 2.2, includes a

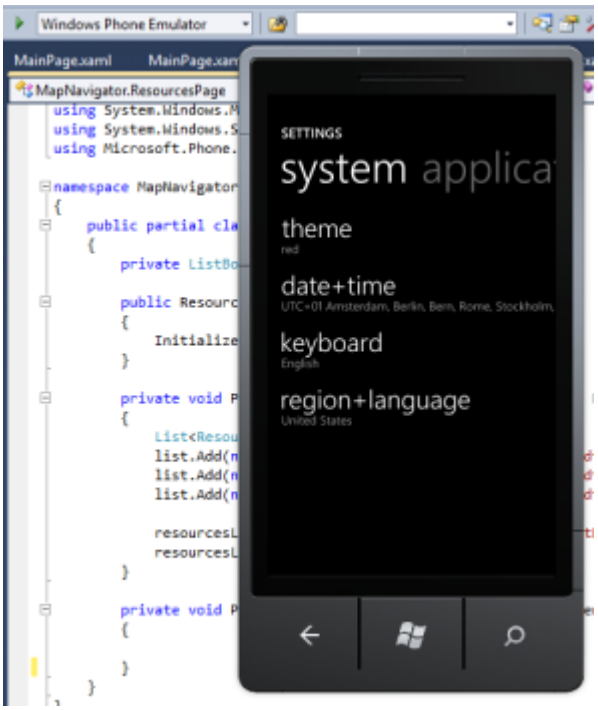


Figure 2.1: WindowsPhone7 software emulator

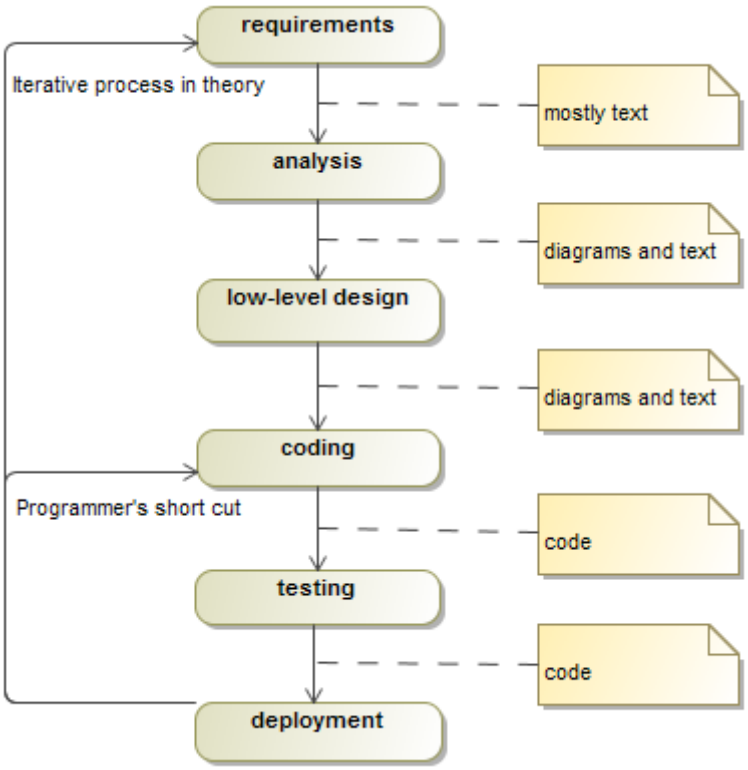


Figure 2.2: Traditional software development life cycle

number of phases:

- Conceptualization and requirements gathering
- Analysis and functional description
- Design
- Coding
- Testing
- Deployment

During phasis 1-3 we produce documents and diagrams that guides the subsequent steps, in an incremental and iterative manner. Changes are often done at the code level only, because the time to update the diagrams and other high-level documents is not available.

2.2.1 The Portability Problem

The first three phases of software development allows us to think the application for a generic mobile device, while coding testing and deploying strictly depend on a specific device. So a long and difficult design work should be forked into three branches of implementation and testing, one for each target platform. In fact, reaching a wide audience of users makes it necessary to develop your application for the most popular operating systems for smartphones on the market, such as Google's Android, Apple's iOS, Nokia's Symbian, Microsoft's Windows Phone and so on. In the next sections we introduce in details two of these smartphone platforms (and briefly discuss others), in order to identify the common traits of the different environments and development approaches.
This analysis will address the study of a development methodology that takes account of these commonalities, focusing on modeling the application independently from the target device and automating the porting to different platforms. The main alternative to this approach to the portability problem, is the use of a cross-platform middleware. The middleware poses itself between the below device platform and the application presented to the user. This involves running the middleware program, that translates and reproduces the platform-independent description of the application, with a potential decrease in performance. This also requires the end user to install the translation software, to run applications. Two examples of these middleware are PhoneGap[1] (open source) and MonoTouch[2] (commercial).

2.3 Android Platform

Android[3] is a Linux-based operating system for mobile devices such as smartphones and tablet computers, and it is developed by the Open Handset Alliance[4] led by Google. Android consists on a kernel based on Linux, a middleware for communications and key applications, including the web browser and the file manager. Developers can download for free the software development kit (SDK), that includes compiler, debugger and the Eclipse plugin to work more efficiently in the Integrated Development Environment. The SDK also provides a software emulator for ARM-based mobile devices. Alternatively, applications can be run directly on the physical device connected via USB cable to the develop machine.

[1] http://phonegap.com/
[2] http://xamarin.com/monotouch
[3] http://www.android.com
[4] http://www.openhandsetalliance.com/

The language used to program Android applications is Java, with a customized class library that bind to some system components (comunication middleware, device camera and audio, ecc). Android uses the Dalvik Virtual Machine with just-in-time compilation to run Dalvik dex-code (Dalvik Executable), which is usually translated from Java bytecode. Each application lives in its own security sandbox (runs on a separated Dalvik Virtual Machine), to secure the system from malicious code. A good starting point to the study of this platform is [2].

2.3.1 Application Structure

The Android SDK tools compile the code, along with any data and resource files, into a package, an archive file with the `.apk` suffix. All the code in a single `.apk` file is considered to be one application and is the file that Android-powered devices use to install the application.

The main component of an android application is the *Activity*, which represents a single screen with an user interface.

A package consists of:

- Class files
  All the java files that program the behavior of application
- Resources files
  An Android application is composed of more than just code. It requires resources that are separated from the source code, such as images and anything relating to the visual presentation of the application. For example, you should define menus and the layout of activity user interfaces with XML files
- XML manifest file
  The application must declare all its components in this file. The manifest file also:
  - identifies any user permissions the application requires
  - declare hardware and software features used or required by the application

With the SDK can be written different kind of applications, such as user applications or services (for instance, a background application that waits for remote connections). We focus our attention on those applications that involve also the design of a Graphical User Interface (GUI).

2.3.2 Activity

An Activity is a basic application component that provides a screen with which users can interact in order to, for instance, dial the phone or send an email.

An activity can exist in essentially three states:

- *Resumed* - The activity is in the foreground of the screen and has user focus
- *Paused* - Another activity is in the foreground and has focus, but this one is still visible
- *Stopped* - The activity is completely obscured by another activity

The complete activity lifecycle schema is depicted in Figure 2.3. If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its finish() method), or simply killing its process.

Activity code and layout are separated into java classes and XML files. To implement an activity, a class that extends `Activity` must be created. This new class should contain a private variable for each widget expected in the screen and override some callback methods that allow the managing of the activity life cycle.

The overridden method `onCreate` is invoked when activity is first created and should initialize the private widget variables by calling the function `FindViewById()`. This function returns a reference to the widget name passed as parameter. Other important overridable methods are:

- onDestroy()
  Called before the activity is destroyed
- onResume
  Called just before the activity starts interacting with the user

Listing 2.1: Sample Activity Class File

```
1 import it.univr.mapnavigator.R;
2 import android.app.Activity;
3 import android.view.View;
4 import android.view.View.OnClickListener;
5 import android.widget.ImageButton;
6
7 public class MainActivity extends Activity implements OnClickListener {
```

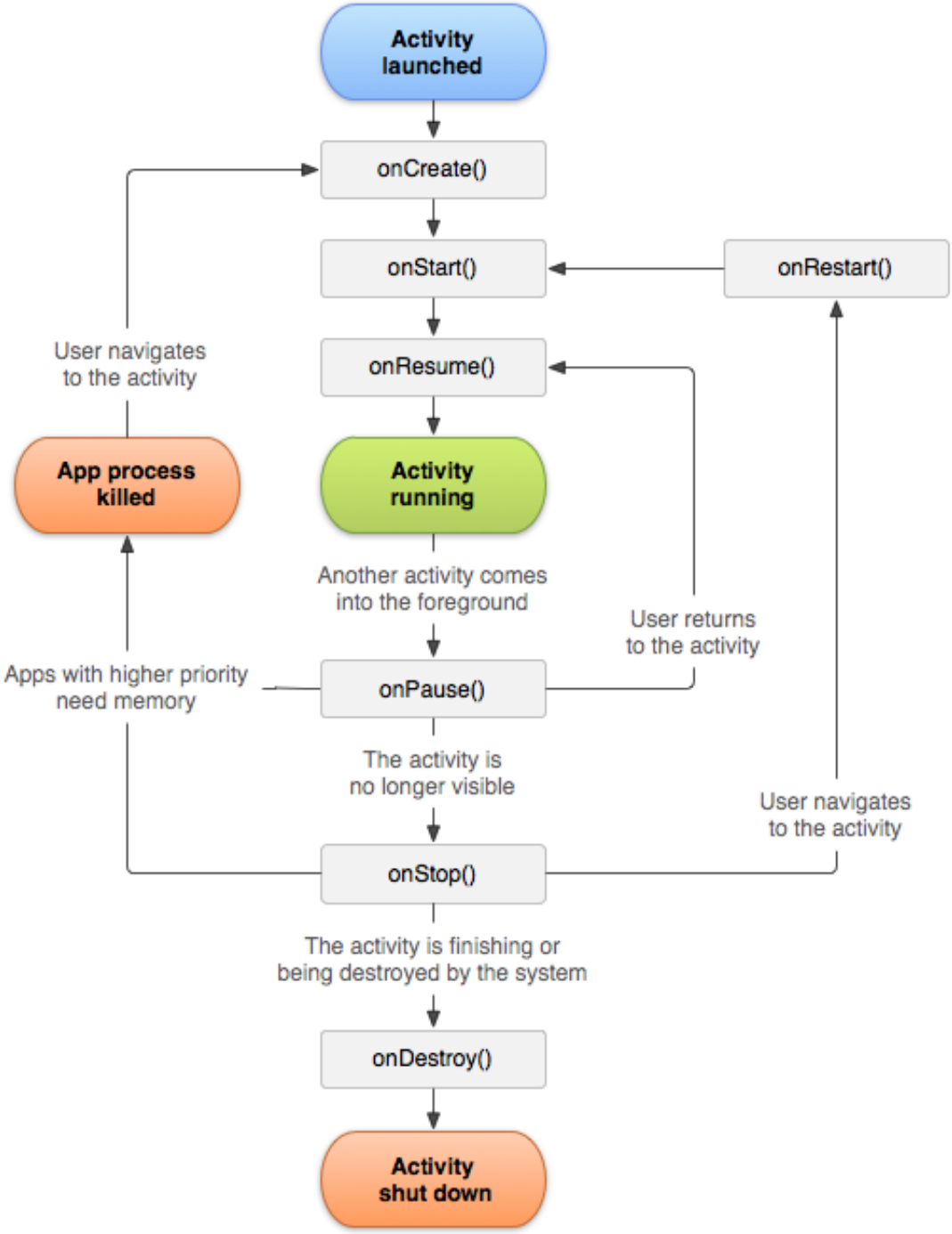


Figure 2.3: The Android Activity lifecycle

```
8   private ImageButton buttonOpen;
9   ...
10
11      @Override
12      public void onCreate(Bundle savedInstanceState) {
13          super.onCreate(savedInstanceState);
14          setContentView(R.layout.mapnav_main);
15          setTitle("MapNavigator");
16          buttonOpen = (ImageButton) findViewById(R.id.imageButtonOpen);
17          buttonOpen.setOnClickListener(this);
18      }
19
20  public void onClick(View v) {
21      switch (v.getId()) {
22        case R.id.imageButtonOpen:
23          ...
24          break;
25      }
26  }
27 }
```

The screen layout is described by an XML file, as the following:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="fill_parent"
4      android:layout_height="fill_parent"
5      android:weightSum="1" android:orientation="vertical">
6      <ImageButton android:layout_height="wrap_content"
7          android:src="@drawable/mappa"
8          android:id="@+id/imageButtonOpen"
9          android:layout_alignParentLeft="true"
10         android:layout_marginLeft="51dp"
11         android:layout_marginTop="28dp">
12   </ImageButton>
13     <TextView android:id="@+id/textView1"
14         android:textAppearance="?android:attr/textAppearanceMedium"
15         android:layout_width="wrap_content"
16         android:text="@string/labelApriMappa"
17         android:layout_toRightOf="@+id/imageButtonOpen"
18         android:layout_marginLeft="18dp"
19   </TextView>
20 </RelativeLayout>
```

2.3.3 Manifest and Resources

Every application must have the `AndroidManifest.xml` file in its root directory. The manifest presents essential information about the application to the Android system, information the system must have before it can run any of the application code. Manifest must contains the following elements:

- the package name that serves as a unique identifier for the application,
- the set of components of the application, such as activities, services and so on,

- permissions the application must have in order to access protected parts of the API and interact with other applications.

Listing 2.2: Sample AndroiManifest.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode=
       "1" android:versionName="1.3" package="it.univr.mapnavigator">
3    <uses-sdk android:minSdkVersion="8" />
4    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
5    <uses-permission android:name="android.permission.CAMERA"/>
6    <uses-permission android:name="android.permission.INTERNET"/>
7    <uses-feature android:name="android.hardware.camera"/>
8    <application android:icon="@drawable/mappa2" android:label="@string/app_name">
9      <activity android:name="mapnavigator.activity.MainActivity" android:label="@string/
           app_name" android:screenOrientation="portrait">
10       <intent-filter>
11         <action android:name="android.intent.action.MAIN" />
12         <category android:name="android.intent.category.LAUNCHER" />
13       </intent-filter>
14     </activity>
15     <activity android:name="mapnavigator.activity.FileChooserActivity" android:label="@string/
           sceglimappa" android:screenOrientation="portrait"/>
16     <activity android:name="mapnavigator.activity.NavigatorActivity" android:launchMode="
           singleTask" android:theme= "@android:style/Theme.NoTitleBar.Fullscreen"></activity>
17     <activity android:name="mapnavigator.activity.CameraActivity" android:screenOrientation
           ="portrait"/>
18   </application>
19 </manifest>
```

Resources are arranged in xml files contained in /res directory. Here's a brief summary of some resource type:

- **Drawable** define various graphics with bitmaps,
- **Layout** define the layout of application UI,
- **Menu** define the contents of application menus,
- **String** define strings and string arrays (and include string formatting and styling).

Android assigns a constant to each resource. This constant allows direct access to the resource within the application code.

2.4 Windows Phone Platform

In 2010 Microsoft presented Windows Phone 7 (WP7 from now on), his new operating system for smartphone. WP7 is radically different from previous

versions, which had a Graphical User Interface modeled on those of desktop versions, not suitable for modern touchscreens. Microsoft, like Google, provides a Software Development Kit to allow developers to design, compile, debug and deploy their own applications. The SDK comes with the Windows Phone Emulator, a desktop application that emulates a Windows Phone device. Emulator provides a virtualized environment in which applications can be debugged and tested without the physical device.

Windows Phone is based on a set of technologies:

- .Net Framework and the main language of the framework, C-Sharp
- Extensible Application Markup Language (XAML)
- Silverlight, the application framework that provides a retained mode[6] graphics system similar to Windows Presentation Foundation[7] (used for desktop GUI), and integrates multimedia, graphics, animations and interactivity into a single runtime environment. In Silverlight applications for Windows Phone, user interface is declared in XAML and programmed using a subset of the .NET Framework.

2.4.1 Application Structure

The structure is similar to that seen in Android. Application code is separated, as usual, from the GUI description, that is coded into XAML files. Resources table is quite different, and can includes only images, strings or generic files embedded in the application package.

2.4.2 Pages

The different screens of the application are represented by a number of *Pages*. Each page is defined by a class that extends *PhoneApplicationPage*. This class is splatted into two partial classes by VisualStudio[8] IDE, hiding to developer some methods and internal variables.

[6]Retained mode rendering is a style for application programming interfaces of graphics libraries, in which the libraries retain a complete model of the objects to be rendered

[7]http://msdn.microsoft.com/en-us/library/aa970268.aspx

[8]Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop console and graphical user interface applications along with Windows Forms applications, web applications, web services and smartphone application in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Phone, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

Listing 2.3: A page class

```
using System;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Phone.Controls;

namespace MapNavigator {
    public partial class MainPage : PhoneApplicationPage {
        public MainPage() {
            InitializeComponent();
        }

        private void buttonOpen_Click(object sender, RoutedEventArgs e) {
            NavigationService.Navigate(new Uri("/ResourcesPage.xaml", UriKind.Relative));
        }
    }
}
```

Listing 2.4: Parzial class hidden to developer

```
namespace MapNavigator {
    public partial class MainPage : Microsoft.Phone.Controls.PhoneApplicationPage {
        internal System.Windows.Controls.Grid LayoutRoot;
        internal System.Windows.Controls.StackPanel TitlePanel;
        internal System.Windows.Controls.TextBlock ApplicationTitle;
        internal System.Windows.Controls.Grid ContentPanel;
        internal System.Windows.Controls.Button buttonOpen;
        internal System.Windows.Controls.TextBlock textBlock1;

        private bool _contentLoaded;

        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent() {
            if (_contentLoaded) return;
            _contentLoaded = true;
            System.Windows.Application.LoadComponent(this, new System.Uri("/MapNavigator;
                component/MainPage.xaml", System.UriKind.Relative));
            this.LayoutRoot = ((System.Windows.Controls.Grid)(this.FindName("LayoutRoot")));
            this.TitlePanel = ((System.Windows.Controls.StackPanel)(this.FindName("TitlePanel")));
            this.ApplicationTitle = ((System.Windows.Controls.TextBlock)(this.FindName("
                ApplicationTitle")));
            this.ContentPanel = ((System.Windows.Controls.Grid)(this.FindName("ContentPanel")));
            this.buttonOpen = ((System.Windows.Controls.Button)(this.FindName("buttonOpen")));
            this.textBlock1 = ((System.Windows.Controls.TextBlock)(this.FindName("textBlock1")));
        }
    }
}
```

Hidden code is auto-generated (before compilation phase) by a tool, based on the description of the GUI Layout. This is the code that at run time initializes GUI components, combined with private local variables that refers to them. The explicit function (like FindViewByID seen in paragraph 2.3.2) to obtain a reference to a GUI component is `FindName()`.

The screen layout is described by an XAML file, as the following:

```
<phone:PhoneApplicationPage
    x:Class="MapNavigator.MainPage"
```

```
3     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
4
5     <!--LayoutRoot is the root grid where all page content is placed-->
6     <Grid x:Name="LayoutRoot" Background="Transparent" ShowGridLines="False">
7         <Grid.RowDefinitions>
8             <RowDefinition Height="Auto"/><RowDefinition Height="*"/>
9         </Grid.RowDefinitions>
10        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
11            <TextBlock x:Name="ApplicationTitle" Text="MAP NAVIGATOR" Style="{
                  StaticResource PhoneTextNormalStyle}"/>
12        </StackPanel>
13        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
14            <Button Height="100" HorizontalAlignment="Left" Margin="70,97,0,0" Name="
                  buttonOpen" VerticalAlignment="Top" Width="100" Click="buttonOpen_Click
                  ">
15                <Button.Background>
16                    <ImageBrush ImageSource="/MapNavigator;component/Images/misc_gift_48.
                          png" />
17                </Button.Background>
18            </Button>
19            <TextBlock Height="30" HorizontalAlignment="Left" Margin="220,134,0,0"
                  Name="textBlock1" Text="Apri_mappa" VerticalAlignment="Top" Width="
                  174" />
20        </Grid>
21    </Grid>
22 </phone:PhoneApplicationPage>
```

2.4.3 Manifest and Security

Every application must have the `AndroidManifest.xml` file in its root directory, and it presents essential information required in application startup phase. Furthermore, when applications are submitted to the Windows Phone Marketplace, information from the manifest file is used in the certification process.

App element (at line 3 in the following example) supplies information such as the product ID, version, and type of application.

Listing 2.5: Sample WMAppManifest.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <Deployment xmlns="http://schemas.microsoft.com/windowsphone/2009/deployment"
       AppPlatformVersion="7.0">
3    <App xmlns="" Title="MapNavigator" RuntimeType="Silverlight" Version="1.0.0.0" Genre
         ="apps.normal"
4    Author="MapNavigator_author" Description="Sample_description" Publisher="MapNavigator"
         >
5      <IconPath IsRelative="true" IsResource="false">ApplicationIcon.png</IconPath>
6      <Capabilities>
7        <Capability Name="ID_CAP_NETWORKING"/>
8        <Capability Name="ID_CAP_WEBBROWSERCOMPONENT"/>
9      </Capabilities>
10     <Tasks>
11       <DefaultTask Name ="_default" NavigationPage="MainPage.xaml"/>
12     </Tasks>
13     <Tokens>
14       <PrimaryToken TokenID="MapNavigatorToken" TaskName="_default">
```

```
15          <TemplateType5>
16            <BackgroundImageURI IsRelative="true" IsResource="false">Background.png</
                  BackgroundImageURI>
17            <Count>0</Count>
18            <Title>MapNavigator</Title>
19          </TemplateType5>
20        </PrimaryToken>
21      </Tokens>
22    </App>
23  </Deployment>
```

Capabilites element, reported in sample code 2.5 al line 6, collects all the permissions that the application needs in order to access the system network stack and run the default web browser.

2.5 Other platforms

Android and Windows Phone, although based on different technologies and languages, are very similar because they shares application structure e development approach. Other platforms adopt a similar development schema, and following are reported some considerations about some of them. iOS (iPhone OS prior to June 2010) is Apple's mobile operating system. Apple does not license iOS for installation on non-Apple hardware, and the official Software Development Kit runs only on Mac OS X operating system. Objective C is the Object Oriented language used to program the applications, while the UI Layout is described separately, according to the schema seen for android and Windows Phone. The scenario of the most recent Symbian platform is instead dominated by the QT[9] library, a cross-platform application and UI framework. Even in this case, the QT framework works taking separate the description of the graphical interface, by binding interactions with the elements that compose the GUI to the application code written in C.

[9]http://qt.nokia.com/products/

Chapter 3

Model Driven Architecture

The Model Driven Architecture[1] (MDA) is a framework for software development defined by the Object Management Group[2]. Key to MDA is the importance of models in the software development process. Within MDA the software development process is driven by the activity of modeling your software system. The MDA development life cycle does not look very different from the traditional one, but the major differences lies in the nature of the artifacts that are created during the developement process. The artifacts are formal models, i.e., models that can be understood by computers. The following three concepts are at the core of the MDA.

Platform Independent Model PIM is a model of software system that is independent of the specific technological platform used to implement it.

Platform Specific Model PSM is a model of a system targeted to a specific execution environment linked to a specific technological platform. This model is indispensable for the actual implementation of a system.

Model transformation For each specific technology platform a separate PSM is generated from the PIM.

MDA therefore divides software development in three main steps:

- creation of the Platform Independent Model,
- transformation of Platform Independent Model in Platform Specific Model,

[1] http://www.omg.org/mda/
[2] http://www.omg.org

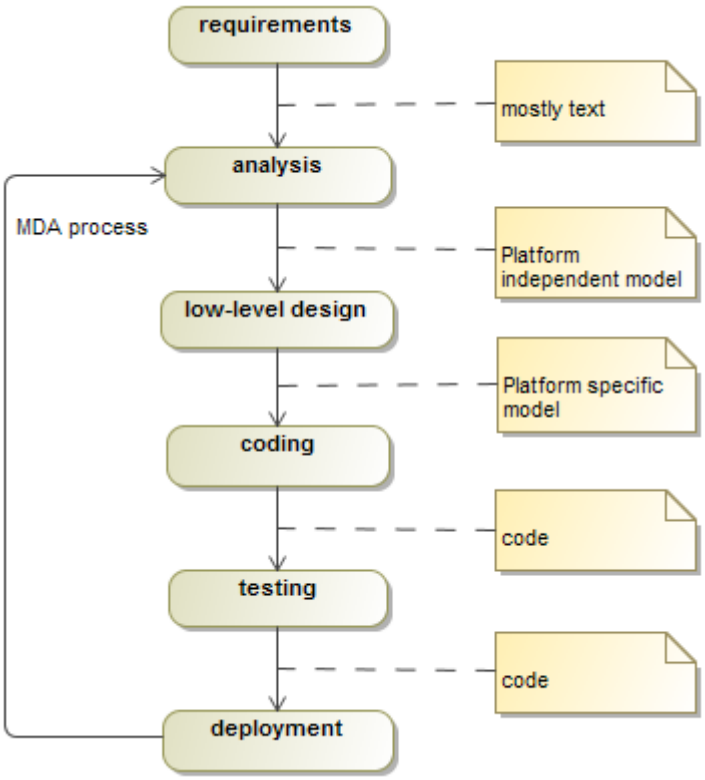


Figure 3.1: MDA software development life cycle

- generation of Software (code).

The MDA's idea borns with the evolution of visual modelization languages, such as UML. MDA is specifically intended to provide a conceptual framework, as comprehensive as possible, for a model-based approach to development. Often the purpose of modeling is simply intended to provide formal documentation of the application and everything that is "not programming" is still seen with suspicion. On the contrary the development by models should allow greater focus on providing solutions to challenging problems of development.

The basis for model development is the Unified Modeling Language, a formal language that can describe "what" makes the application, even before understanding "how" it does.

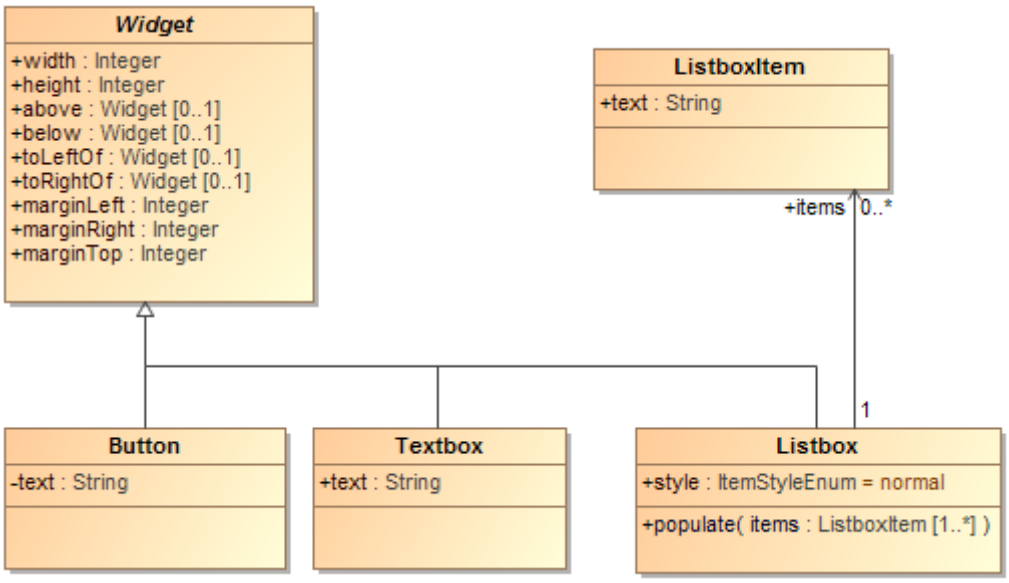


Figure 3.2: A class diagram

3.1 Unified Modeling Language

UML is a standardized general-purpose modeling language consisting of a family of graphical notations, that help to describe and to design software systems. UML diagrams represent two different views of a system model, static and dynamic, that covers structural and behavioral aspects.

Follows a brief description of the major and useful diagrams. For a complete discussion of Unified Modeling Language see *UML Distilled* [3] and *UML explained* [4].

3.1.1 Structural Diagrams

Structural diagrams show static structure of the system and its parts on different abstraction and implementation levels, and how those parts are related to each other. The elements in a structural diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.

Class Diagram shows structure of the designed system at the level of classes and interfaces, shows their features, constraints and relationships - associations, generalizations, dependencies, etc.

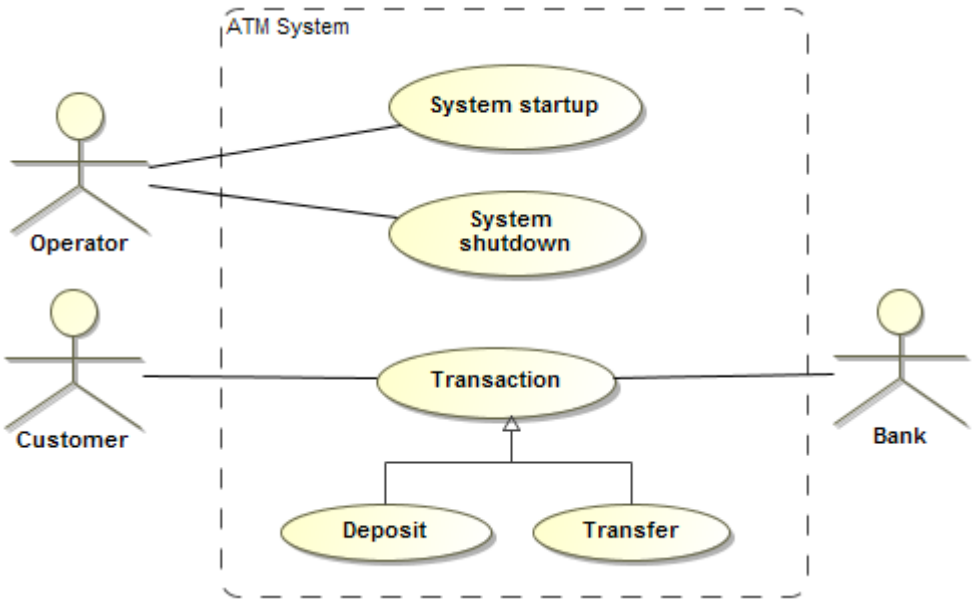


Figure 3.3: An use case diagram

Object Diagram is a graph of instances, including objects and data values. An object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time

Component Diagram shows components and dependencies between them. These diagrams are useful when we divide the system in components and want to show their interrelationships trough interfaces.

3.1.2 Behavioral Diagrams

Behavioral diagrams depict behavioral features of a system or business process. They capture the varieties of interaction and instantaneous state within a model as it "executes" over time.

Use Case Diagrams are a technique for capturing the functional requirements of a system. They can be used to describe a set of actions (use cases) that some systems should or can perform in collaboration with one or more external users (actors).

State Machine Diagram shows discrete behavior of a part of designed system through finite state transitions. Behavior is modeled as a traversal of

a graph of state nodes connected with transitions. Transitions are triggered by the dispatching of series of events and, during the traversal, the state machine could also execute some activities.

Activity Diagram shows flow of control or object flow with emphasis on the sequence and conditions of the flow. The actions coordinated by activity models can be initiated because other actions finish executing or because objects and data become available.

Sequence Diagram shows how processes operate with one another and in what order. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams typically are associated with use case realizations in the Logical View of the system under development.

3.2 Object Constraint Language

Object Constraint Language (OCL) is an expression language that enables one to describe constraints on object-oriented models and other object modeling artifacts. UML has some weak points in the behavioral or dynamic part. It includes many different diagrams to model dynamics, but their definition is not formal and complete enough to describe how to transform a PIM in a PSM and cover all behavioral aspects in a precise manner. OCL is a precise text language that provides constraints and object query expressions on any MOF model or meta-model that cannot otherwise be expressed by diagrams. OCL can also used for the following purposes:

- Specifying initial attribute values
- Specifying guard conditions in statecharts
- Specifying constraints between elements of different diagrams
- Specifying the derivation rules for attributes or associations

For more details on OCL language refer to *The Object Constraint Language, Precise Modeling with UML* [5].

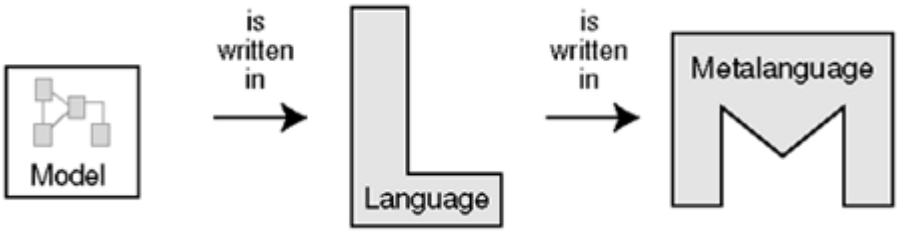


Figure 3.4: Models, languages and metalanguages

3.3 Models and Metamodeling

A model is a description of a system written in a well-defined language, that is a language suitable for automated interpretation by a computer. Because modeling languages do not have to be text based (they can have a graphical syntax, as we have seen in section 3.1), it is necessary a different mechanism for defining languages in the MDA context. This mechanism is called *metamodeling*, and this argument is discussed in Chapter 8 in [6].
Every kind of element that a modeler can use in his model is defined by the metamodel of the language the modeler uses. Because a metamodel is also a model, metamodel itself must be written in a well-defined language. This language is called *metalanguage*. Backus-Naur Form (BNF), a grammar to define languages, is an instance of metalanguage.

In MDA framework a metalanguage plays a different role than a modeling language, because it is a specialized language to describe modeling language. The metamodel completely defines the language. Figure 3.4 shows the relationship between a model, its language and the metalanguage.

3.4 OMG's Standards for Metamodeling

As seen in the previous section, a metalanguage is a language itself and it can be defined by a metamodel written in another metalanguage. In theory there is an infinite number of layers of model-language-metalanguage relationships. The standards defined by the OMG use four layers.

Layer M0: The instances At the M0 layer there is the running system in which the actual ("real") instances exist.

Layer M1: The Model of the System This layer contains, for example, a UML model of the system and defines concepts like *Customer* with the properties *name, street, city* There is a relationship between the M0 and M1 layers, and the concepts at the M1 layer are all classifications of instances at the M0 layer.

Layer M2: Model of the Model The elements that exist at the M1 layer (classes, attributes, and other model elements) are themselves instances of classes at M2, the next higher layer. An element at the M2 layer specifies the elements at the M1 layer. The model that resides at the M2 layer is called a *metamodel*. Every UML model at the M1 layer is an instance of the UML metamodel as defined in UML 2.0 Specification [7].

Layer M3: The Model of M2 This is a metameta layer and between elements of M2 and M3 exists the same relationship that is present between elements of layers M0 and M1, and elements of layers M1 and M2. Layer M3 defines the concepts needed to reason about concepts from layer M2. MetaObject Facility[3] (MOF) is the standard M3 language defined by OMG. All modeling languages (like UML) are instances of the MOF.

Figure 3.5 summarizes the relationships between the layers.

Another important OMG's standard is XML Metadata Interchange (XMI) that can be used to exchange any metadata whose metamodel can be expressed in MOF. The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages (metamodels).

3.5 Model Transformations

A transformation is the generation of a target model from a source model. The process is described by a *transformation definition*, which consists of a number of *transformation rules*, and is executed by a *transformation tool*. A transformation rule maps an element from the source model to the specific one (or more than one) in the target model. It also can add attributes or relations between elements in the target model, depending on the purpose of the transformation.

[3] http://www.omg.org/mof/

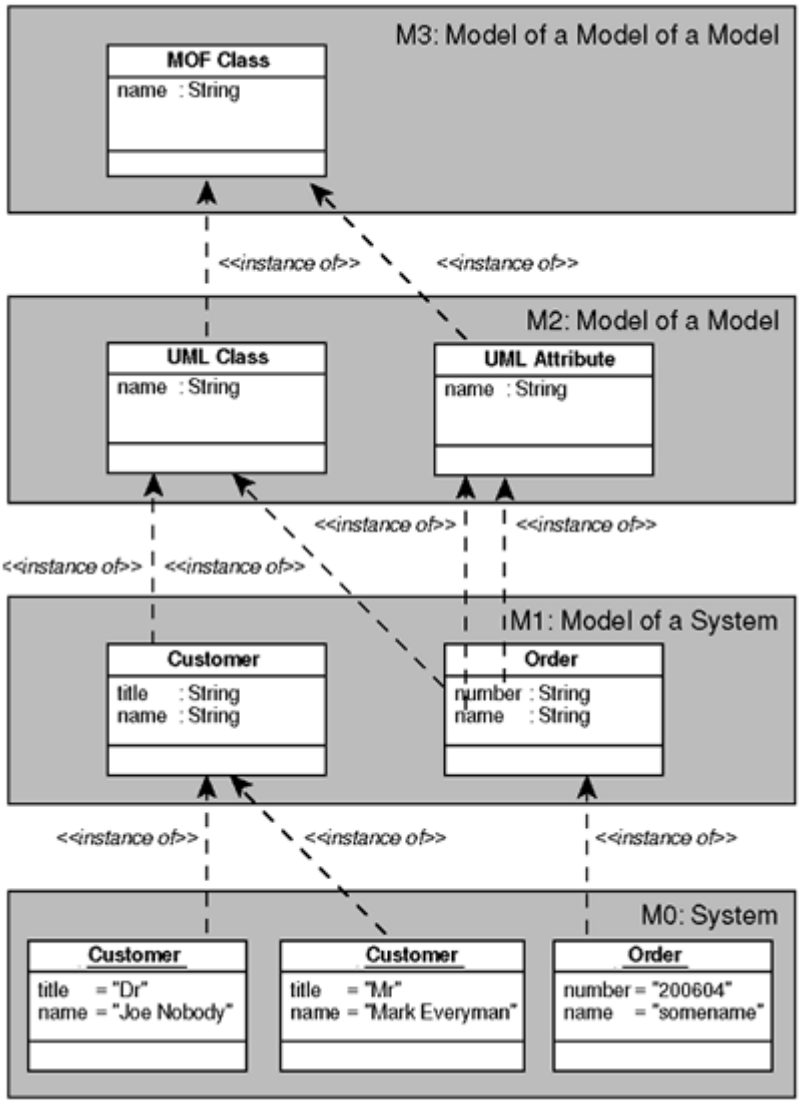


Figure 3.5: Overview of layers M0 to M3

In MDA approach there are a number of features of the transformations process that are desirable:

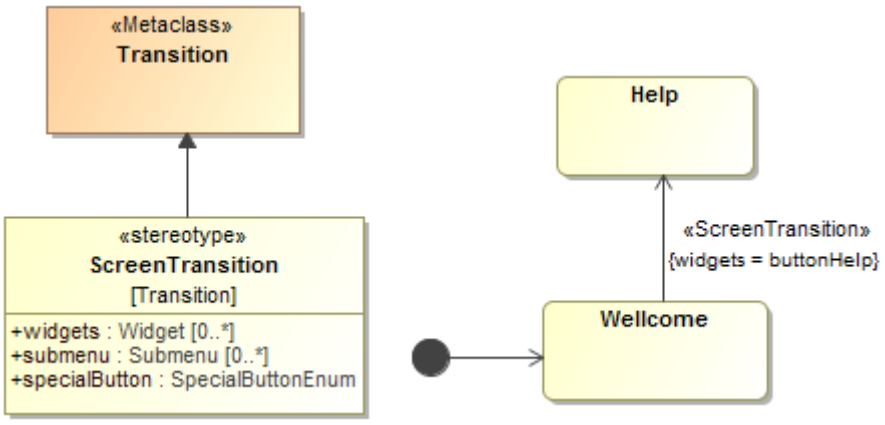
- **Traceability**, which means that an element in the target model must be traced back to the element(s) in the source model from which it is generated
- **Incremental consistency**, which means that when target-specific information has been added to the target model and it is regenerated, the extra information persists.
- **Bidirectionality**, which means that a transformation can be applied not only from source to target, but also back from target to source.

3.6 The UML Extension Mechanisms

UML provides a lightweight extension mechanism by defining custom *stereotypes, tagged values* and *constraints*. UML Profiles Diagrams allow adaptation of the UML metamodels for different platforms or domains. For example, semantics of standard UML metamodel elements could be specialized in a profile. In a model with the profile "Java model" generalization of classes should be able to be restricted to single inheritance.
The profiles mechanism is not a first-class extension mechanism. It does not allow to modify existing metamodels or to create a new metamodel as MOF does. Profile only allows adaptation or customization of an existing metamodel with constructs that are specific to a particular domain, platform, or method.

Profiles can be dynamically applied to a model and in general is characterized by one or more of the following elements:

- rules that specify when a model of the profile should be considered "well formed". These rules take the form of additional constraints (respects to those defined by the UML standard) which restrict the possibilities of use and composition of the language elements,
- stereotypes, tagged value and constraints in addition to those present in the UML standard,
- additional semantic information (natural language) for the elements of UML which is permitted in the context of the profile,
- a set of default model elements, instance of the UML standard constructs.



(a) (b)

Figure 3.6: Stereotypes definition and application on the *Transition* element of the Statechart Diagram

3.6.1 Stereotypes and Tagged Values

A stereotype is an extension of a standard language construct. Stereotypes are one of the basic mechanisms to define specialized versions of UML for particular contexts. In particular it defines how an existing metaclass may be extended as part of a profile. They appear in UML diagrams as text labels in angle brackets, as shown in Figure 3.6.

Chapter 4

Goals

This thesis is focused on the study of a Model-Driven approach to the smartphone applications design. The analysis outlined in chapters 1 and 2 has led to highlight the commonalities between several smartphone software platforms. Establishing a development methodology based on Model Driven Design involves the following goals.

1. Definition of a UML profile for the creation of the Platform Independent Models of smartphone applications, according to the Model Driven Architecture. Using elements and diagrams of UML as metamodel, it is possible to give a representation of the application structure and behavior, thus creating a model that is independent from any specific platform. Profile is needed to provide a set of classes, constraints, and relations to describe structural (for example, GUI layout) and behavioral (transitions between application screens) aspects. All these elements are described by UML metamodel, and represent an abstraction of all the structural and behavioral elements that can be found on all platforms.

2. Detailed description of model transformations to obtain Platform Specific Models. One transformation is needed for each target platform, and this step is crucial to get an automated procedure implementable in a tool. A transformation defines the rules to map a platform independent element (for instance, a widget class provided by UML profile) in the specific one in the target platform.
   PIM and PSM are not only a representation of the application structure (screens, widgets and GUI layout), but also a basic UML model that can be developed and integrated, like in a typical UML design. So, developer can insert in PIM (as well as in PSM) other classes, at-

tributes and methods useful for the application functionality and logic. All these classes should be reported, by the transformation, in PSM.

3. Generation of the application code from the Platform Specific Model. Many UML tools already allow code generation from class diagrams, but in this case the task is more complex, because we should generate code from behavioral diagrams. Furthermore, the code to be generated concerns not only methods signature and class structure, but also blocks of code that implement specific functionality, such as screen transitions. These blocks can be generated by parameterized templates of code.

4. Quality assessment of a sample application obtained by model transformations and code generation, by comparison with the same application realized via traditional (and totally manual) development approach. To achieve this purpose it is necessary to define some evaluation metrics, highlighting the pros and cons of the proposed methodology.

Chapter 5

Proposed Methodology

This chapter introduces a methodology to implement the Model Driven Design approach to the development of smartphone applications. In particular the aim is to design the structural aspect of the application, such as the Graphical User Interface and the structure of classes that govern the application logic, and the behavioral aspect, namely the management of transitions between the screens of the GUI in response to user interaction and system events.
This design must be done thinking independently by a particular software platform, reasoning with the elements and characteristics of smartphone applications. The subsequent transformation from the abstract model to the various specific models for each target platform must be implementable by an automated tool. Finally, this tool must be able to generate the application code (classes, method signatures and some blocks of code), avoiding the developer to write repetitive code.
The proposed methodology refers to the development of the most common type of smartphone applications. Typically "apps" involves a number of screens, composed by a set of witgets, with which users can interact in order to perform different tasks. Other applications that require advanced graphics or customized widgets, like games, cannot be abstracted from the specific target platform. These applications are not covered in this discussion because of their specific nature.

With regard to extensibility and flexibility, the Model-Driven Design is not a monolithic process. It is separated in three main steps, each of which can be taken as a base to integrate and reuse external code and models. Indeed, at each level (platform independent, platform specific, and application code) it is possible to extend the application model with previously designed classes, or to integrate the generated application code with other code already devel-

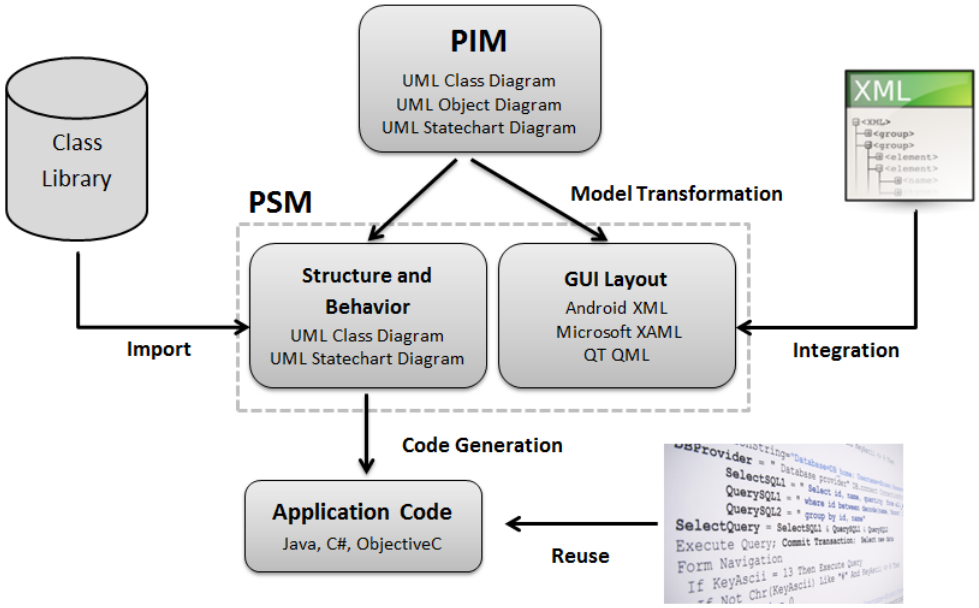


Figure 5.1: Block diagram of the proposed methodology

oped. This is especially useful in PSM, where we can model class libraries for specific and complex functionality that the application can take advantage. In this way it is also possible, at the platform-specific level, to reuse parts of an application previously modeled for a specific platform, for example, the GUI Layout of some Windows Phone Pages. The process is therefore not stand-alone, but it allows the integration with what already exists and that has not been previously modeled thinking independently from a specific platform. Figure 5.1 schematically summarizes this scenerario.

5.1 Platform Independent Model

The first step is to define the language, that is the metamodel, with which to describe the PIM. One possibility is to realize a specific metamodel to "talk about" elements and concepts of the application domain. Instead the choice was to adopt the whole UML metamodel [1] and specify a Profile Diagram to provide classes, interfaces and other artifacts to describe the Platform Independent Model. This choice was made necessary because PIM is not just a representation of the structure of the application, but also a basic

[1] http://www.omg.org/technology/documents/modeling_spec_catalog.htm

model that developers can integrate with other classes or methods, like in a typical UML design.

The UML Profile for Platform Independent Model provides the following elements:

- A list of *Widget Classes*, each of which represents a particular Widget that can be found in every smartphone platform. All these classes extends an abstract class `Widget`, that has a set of attribure that describe the position on the screen. Developer can specializes a widget class in order to design a custom widget class.
- A class `Screen`, that represents a single screen of the GUI
- `Menu` and `Submenu` classes. Menu, indeed, are not considered widgets because they cannot be placed freely in the screen, but typically they are managed directly by the system.
- A stereotype `ScreenTransition` applicable to the element `Transition` of the UML metamodel in a StateChart Diagram.
- A list of *Enumerations* that represent user-defined data types. They are useful for those attributes which are bound to particular and significant values.

Figure 5.2 depicts the entire proposed UML Profile.
The next sections discuss the use of three UML diagrams, each of which constitutes a fundamental element in the model to represent structural and behavioral aspects of the application.

5.1.1 Class Diagram

Class Diagram defines the structure of the smartphone application. A number of classes represent the basic elements of Graphical User Interface, such as screens, widgets and menu. Other classes collect information about application (title, permissions, version) and the abstract device. Table A.1 summarizes the classes provided by Profile.
For each screen, a class that extends the abstract class *Screen*, must be added to the model. The following attributes characterize an application screen:

- `widgets` collects all the widgets that make up the GUI
- `orientation` specifies screen orientation (landscape or portrait)

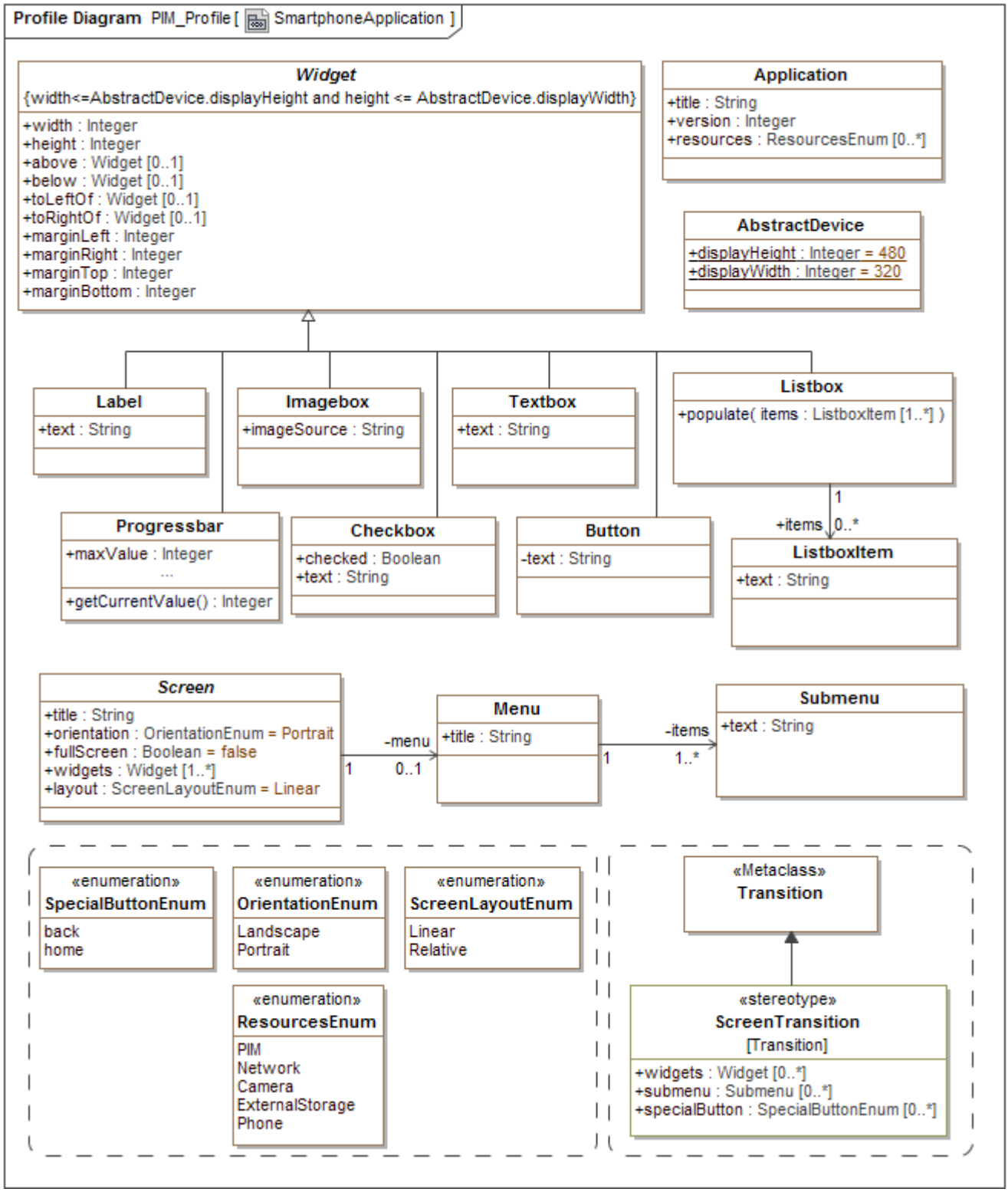


Figure 5.2: UML Profile for Platform Independent Model

- `fullScreen` specifies whether the screen should cover the system menu and the entire display
- `layout` specifies the layout of the widgets on the screen. A Linear layout places each widgets in a new line, instead Relative layout places widgets relative to the others that are around. Section 5.1.3 clarifies how to manage GUI Layout.
- `menu` links the current screen to an instance of the class `Menu`, that represents a Menu and its submenu.

The class `Application` collects, in addition to general information, the list of resources that the application needs to access. Device resources are quite standard on every platform, and refer to:

- *Personal Information Manager* - information like messages, personal contacts, calendar and so on
- *Network communication* - the protocol stacks, such as IP or Bluetooth
- *Device Camera* - the integrated camera that allows the user to take photos and record video
- *ExternalStorage* - additional memory cards for expand device storage capacity
- *Phone* - outgoing and incoming phone calls

The developer can treat the PIM as is usual in UML design. Other custom classes, not falling in those supplied by the Profile, can enrich the diagram. These classes do not represent domain-specific elements, but contribute to enrich the model in order to design the application features and logic.

5.1.2 Enumerations

In the UML profile there are four Enumerations, useful for those attributes which are bound to particular and significant values:

1. `OrientationEnum [Landscape, Portrait]` refer to the orientation of the device
2. `ResourcesEnum [PIM, Network, Camera, ExternalStorage]` lists the resources that the application needs to access.

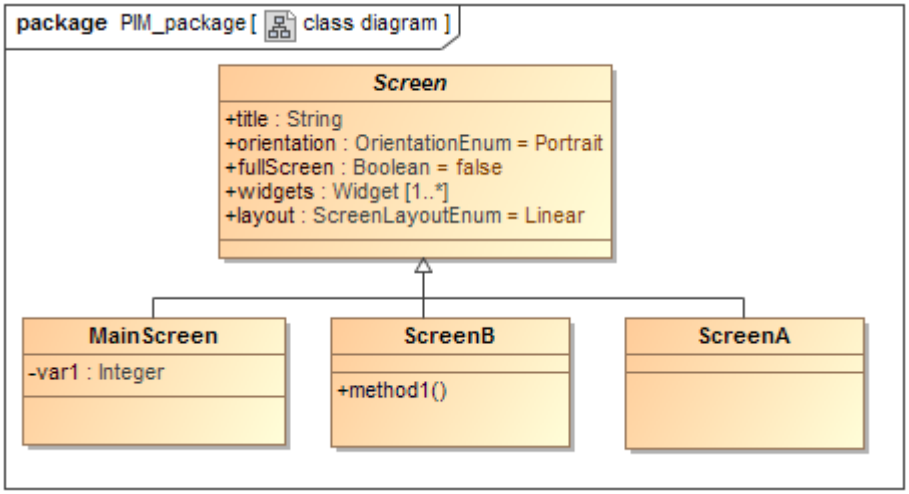


Figure 5.3: Sample Class Diagram for application structure

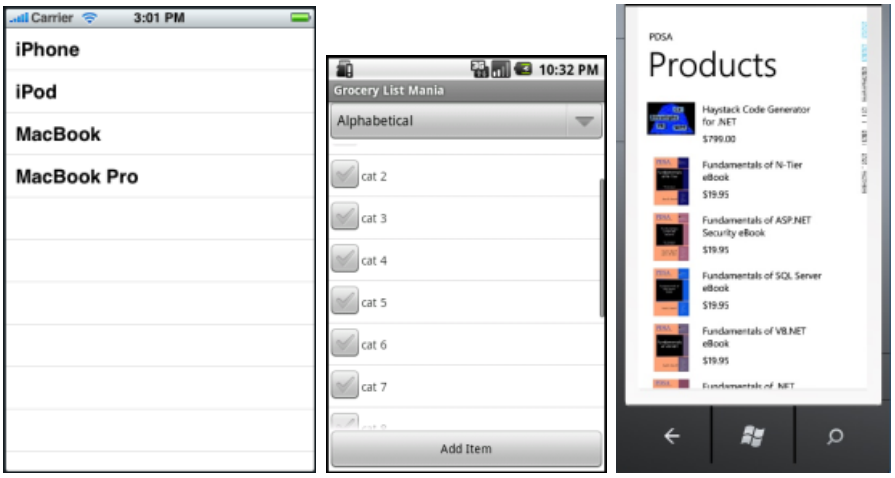
3. `ScreenLayoutEnum [Linear, Relative]` indicates the layout of the widgets in a screen.

4. `SpecialButtonEnum [back, home]` is used by the stereotype ScreenTransition in StateChart Diagram, explained in section 5.1.4

5.1.3 Object Diagram and GUI Layout

Graphical User Interface is a critical aspect of smartphone application, and implementing a model that is independent of a specific platform is an hard task. The proposed UML profile comes with a set of widget classes that can be found in all the major smartphone platforms. Classic widgets can be divided in command widget (Button or ImageButton), input widget (Textbox, Textarea), output widget (Label, Imagebox), structured widget (Listbox) and other widgets like Checkbox and Progressbar. Each platform also takes custom widgets that cannot be generalized. This is a limitation to the design of the GUI, but it is a good starting point to generate the graphical interface structure for each target platform. Subsequently, it can be expanded and adapted to specific needs (whenever custom widgets are needed), but having already designed and reasoned about its features. Figure 5.4 shows a sample implementation, on different platforms, of the Listbox widget.

The proposed Platform Independent Model of Screen GUI adopts two layout type:



(a) (b) (c)

Figure 5.4: The Listbox widget in iOS, Android and WindowsPhone

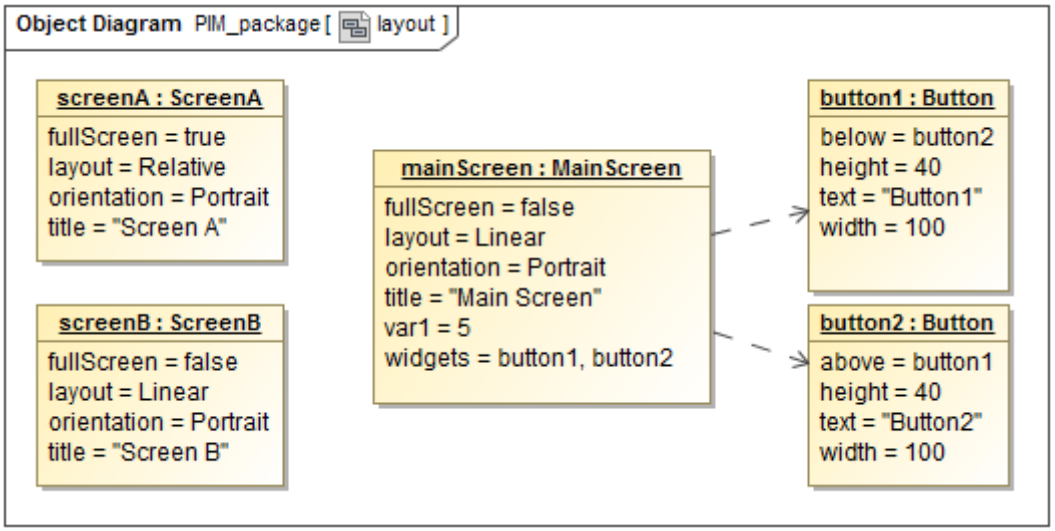


Figure 5.5: Sample Object Diagram for GUI Layout

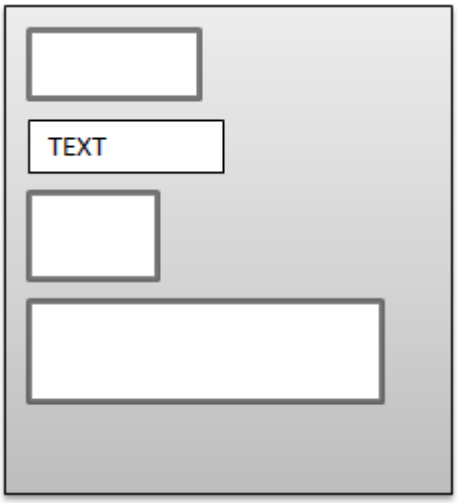


Figure 5.6: Linear Layout

- The *relative layout* in which each widget can specifies the other widgets that has around itself (above, below, right, left). The attributes `marginLeft`, `marginRight`, `marginTop` and `marginBottom` are the only ones that must be obligatorily specified, and they represent the distances between the widgets. So, a widget can be placed specifing its distance from the top-left corner of the screen, leaving empty `above`, `bottom`, `toLeftOf` and `toRightOf` attributes, or specifing the distance from other widgets.

- The simple *Linear layout*, that places all widgets one below the others, according to the order in which they appear in the attribute `widget` of the class `Screen`

In *Relative layout* the following two recursive functions can be used to compute, from the PIM layout representation, the absolute coordinates of the top-left corner of a widget:

$$getX(w) = \begin{cases} w.\text{marginLeft} & \text{if } w.\text{toRightOf undefined} \\ w.\text{marginLeft} + getX(w.\text{toRightOf}) & \text{otherwise} \end{cases}$$

$$getY(w) = \begin{cases} w.\text{marginTop} & \text{if } w.\text{above undefined} \\ w.\text{marginTop} + getY(w.\text{above}) & \text{otherwise} \end{cases}$$

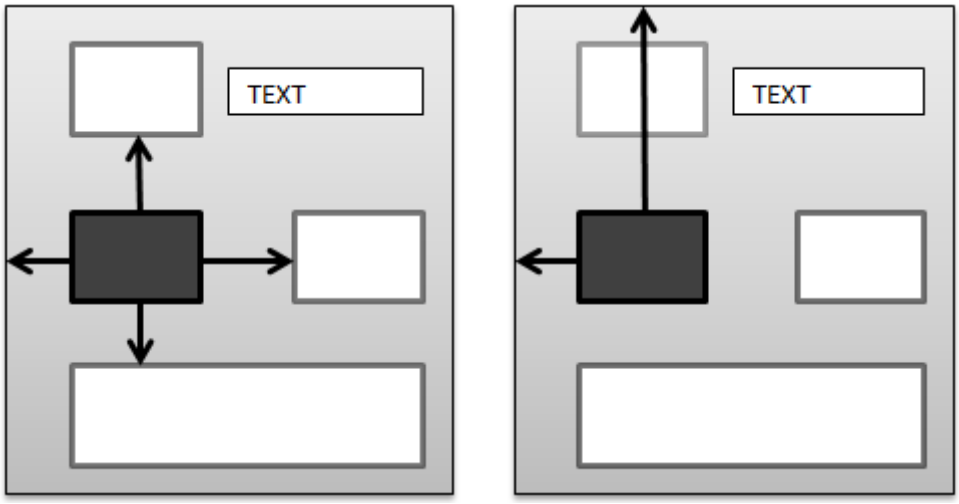


Figure 5.7: Relative Layout

Different smartphones can have different screen resolutions. In the PIM it is necessary to choose a reference resolution and a reference screen density. A good choise is HVGA resolution (320x480) at 160pdi, adopted by several entry-level and mid-level smartphones. Other smartphones can have a higher density, e.g. 240dpi. Density-independent pixel (dp) is virtual pixel unit that can be used when defining UI layout, to express layout dimensions or position in a density-independent way. The density-independent pixel is equivalent to one physical pixel on a 160 dpi screen, which is the baseline density assumed for a "medium" density screen. The conversion of dp units to screen pixels is simple: $px = dp * (dpi/160)$. All the coordinates used in Platform Independent Model are expressed in term of density-pixel unit, and refer to the HVGA screen resolution and to the screen density of 160dpi.

To describe the screen layout it is necessary to instantiate *screen* and *widget* classes, and to set the attributes to the appropriate values. To do so, **UML Object Diagram** can be used to describe relationships between screens and widgets. Once specialized the screen class and selected the widgets composing the GUI, the corresponding widget classes must be instantiated with the appropriate attributes values.
To make the diagram suitable for processing by automated rules, certain conventions must be respected.

1. The Instance Specification of each class specializing the class `Screen` must be linked, by the *Dependency Association*,

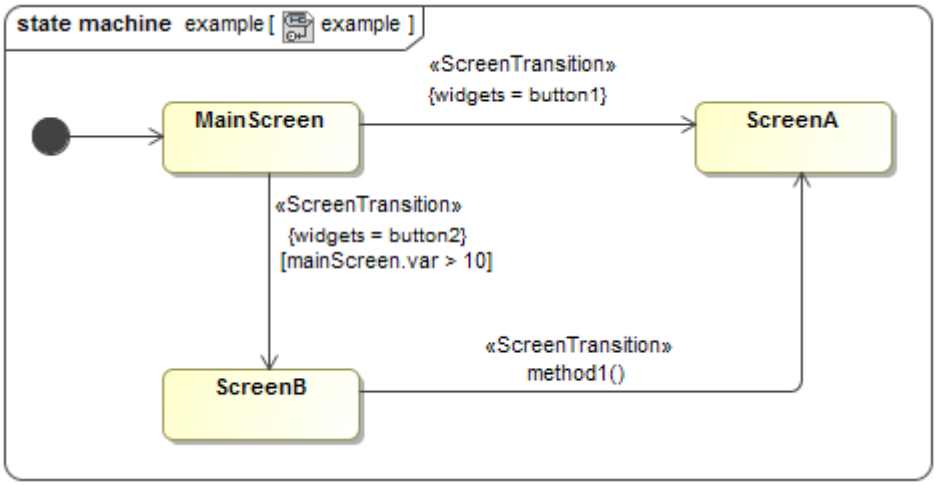


Figure 5.8: Sample Statechart Diagram

- to each Instance Specification of widgets composing the UI of the screen
- to the Instante Specification of the menu (if there is one) composing the UI of the screen

2. Each Instance Specification of a `Submenu` belonging to a `Menu` M, must be linked by the *Dependency Association* to the Instance Specification of M.

3. Each Instance Specification of a `ListboxItem` belonging to a `Listbox` L, must be linked by the *Dependency Association* to the Instance Specification of L.

4. No other *Dependency Association* must be used

5.1.4 Statechart Diagram and Screen Transitions

Once defined the structure of the application, it is usefull to describe the interactions that the user can done with the Graphical User Interface. Every application have a main screen, which is presented to the user when it launches the application for the first time. Next, user can perform a task that involves to interact with many different screens.
The transition between two screens can happen as a result of certain events:

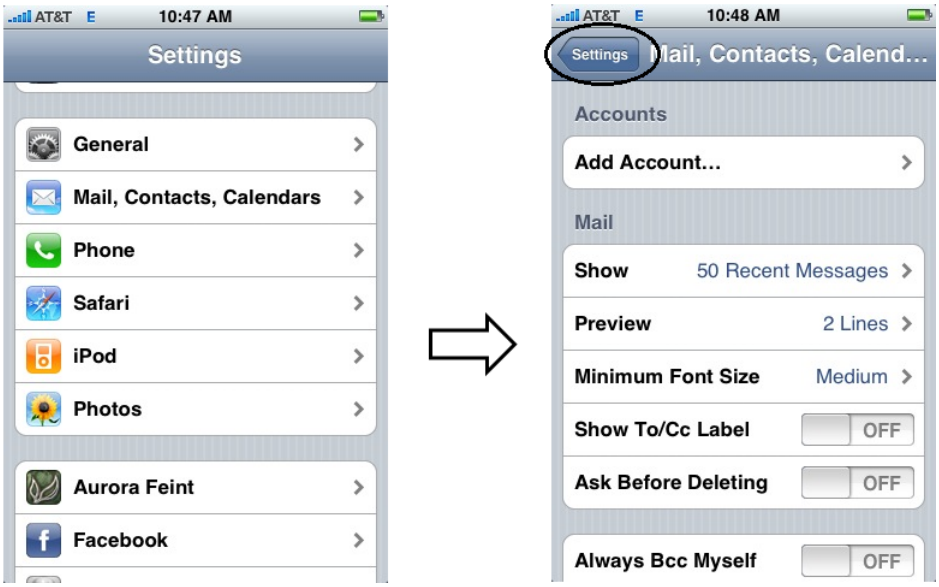


Figure 5.9: *Back button* software implementation on iOS

1. A *Click* or *Touch* event raised by a widget. The user interacts with a widget on the current screen. Each widget can react to the user's "touch" action

2. Invocation of a particular class method that explicitly performs the transition.

3. The user presses a special device button, like "Home" or "Back" button. Almost all smartphones has two special button: *Home* to put the application in background and to show the main system screen, and *Back* to return to the previous screen. iPhone is a special case. It has only one button, a clean and minimalist style. Nevertheless, the *Back* button is typically implemented in the GUI, as shown in figure 5.9.

4. A *Click* or *Touch* event raised by a submenu of a screen menu.

Furthermore, whenever a system event happens, the application can react performing a specific action. Common system events are, for instance, the low battery alert or incoming call alert. Usually these events are managed by an event handler, and they fall in the second case of the previous list.

UML Statechart Diagram can be used to model Screens and their Transitions. It is a representation of a Finite State Machine where:

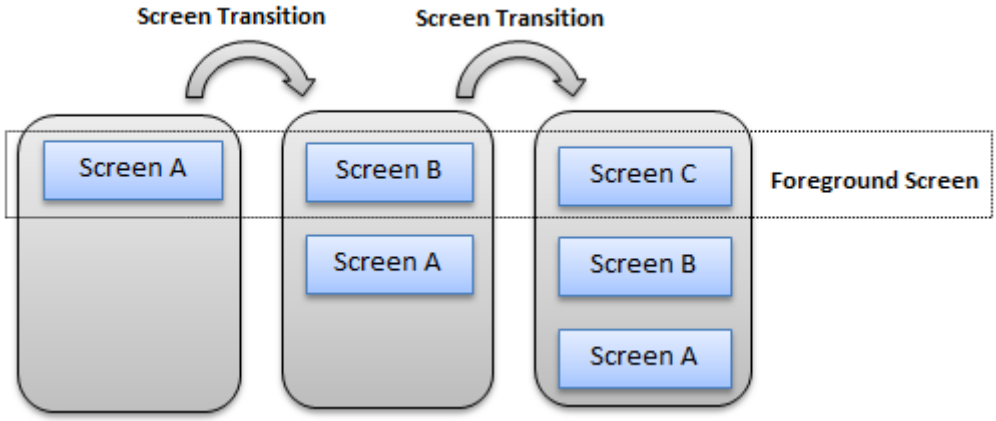
- a state is the union of all configurations of a screen class,
- the initial state is the main Screen of the application,
- the oriented arc from state A to state B represents the transition from screen A to screen B,
- the input are the union of current-screen class methods, current-screen widgets and special buttons,
- no output is defined,
- the set of final states are not defined.

The sequence of screen transitions can be seen as a stack. Whenever the user performs a screen transition, the system puts the target screen on the stack, as shown in figure 5.10. An implicit oriented arc between states A and B, labeled with "Back button" input, is added whenever an oriented arc exists between states B and A. The default behavior of *Back* button removes from the stack che current screen and put in foreground the previous screen. Developer can alter the default behavior intercepting the pressure of the back button, to performing a different screen transition. In this case, an oriented arc, labeled with "Back button" input, must be explicitly added to the graph. Figure 5.11 shows the stack modifications after *Back button* redirection.

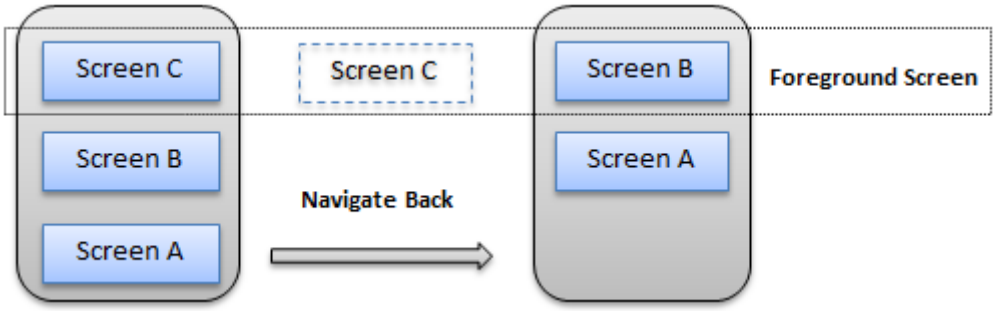
UML Statechart Diagram allows developer to set, on the transition, the name of the method that triggers the transition itself. Instead stereotype *ScreenTransition* should be applied to the UML Transition element, in order to specify whenever the transition happens after pressing a special button or after user interaction with a widget. To do so, Stereotype *ScreenTransition* provides three tagged values:

- widgets: Widget[0..*]
- submenu: [Submenu [0..*]
- specialButton: SpecialButtonEnum [0..1]

Transition *guard* can be expressed by OCL constraint. The context of OCL expressions used in Statechart Diagram must be the package that contains all elements of the PIM. So, expressions can refer to public variables and methods of all classes in the model. The discussion about the translation of OCL boolean expression into java (CSharp, ObjectiveC) code is beyond the scope of this thesis and, in the section illustrating the Code Generation, the details will be omitted.



(a)



(b)

Figure 5.10: Screens Stack and *Back button* default behavior

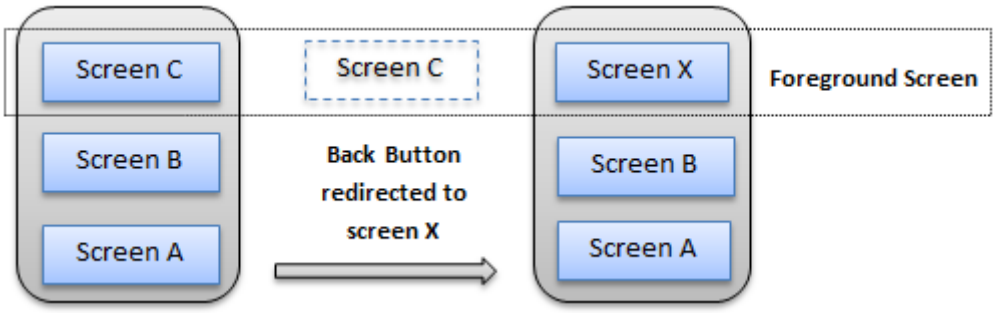


Figure 5.11: Screens Stack and *Back button* redirection

Figure 5.8 reports a sample Statechart diagram with three state. Transition from MainScreen to ScreenA fires after touch event on widget `button1`. Transition from MainScreen to ScreenB fires after touch event on widget `button2`, only if the OCL expression `'mainScreen.var > 10'` holds. Finally, transition from ScreenB to ScreenA fires after the class method invocation.

5.2 Platform Specific Model

The metamodel, or equivalently the language, chosen to describe the Platform Specific Model is once again UML. But it is not the only, because in PSM it is necessary to separate GUI layout from the other parts of application structure. Indeed, every platform adopts its own language to represent screens layout, typically a markup language such as XML (eXtensible Markup Language). Figure 5.1 highlights this separation and reports the diagrams used to describe the application structure and application behavior (Class Diagram and Statechart Diagram), and the languages to describe GUI layout, one for each platform. These markup languages may be considered metalanguages, as seen in section 3.3. For example, *Xaml Object Mapping Specification* [8] reports the metamodel specification of XAML (eXtensible Application Markup Language), used in WindowsPhone platform. PSM must also includes the model, represented by the appropriate and specific metalanguage, of the manifest file, where required.
Another UML Profile must be defined for each target platform, in order to provide the fundamental platform-specific set of classes and interfaces, the stereotype for transitions in UML Statechart Diagram and the appropriate OCL contraints. Figures 5.12 and 5.12 depict respectively Android and WindowsPhone UML Profile. For convenience, in the figure were not reported all attributes and all methods of the classes, but only those most significant.

5.3 Transformations and Rules

Model transformations represent the core of Model-Driven Design. A transformation consists of a set of rules that map a source model element in the specific one (or more than one) of the target model, and should meet the three principles seen in section 3.5: traceability, incremental consistency and bidirectionality. The main purpose of the next two sections is the definition of the set of rules that carry out the transformations, in order to make it possible the algorithmic implementation of the transformations. The rules

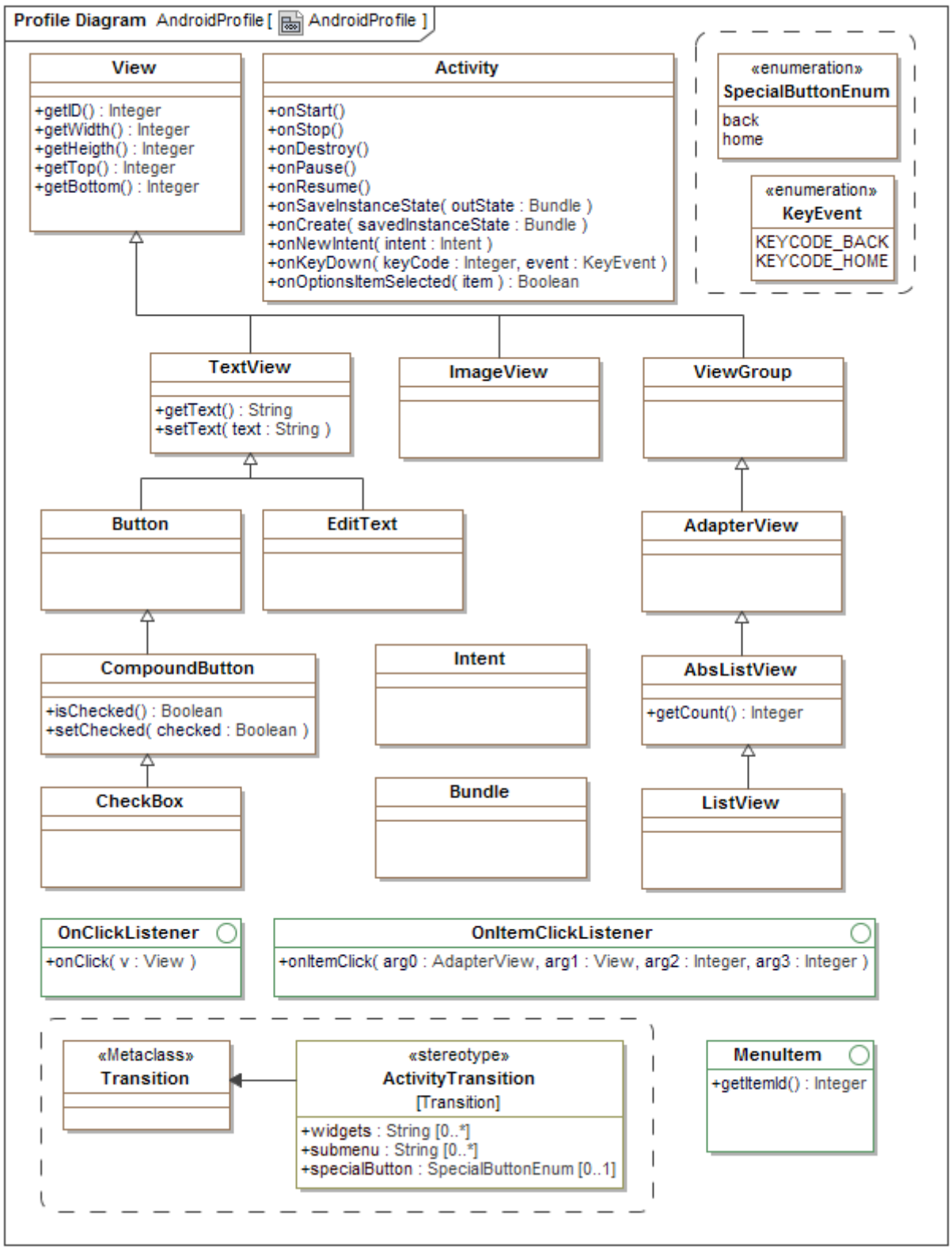


Figure 5.12: UML Profile for Android PSM

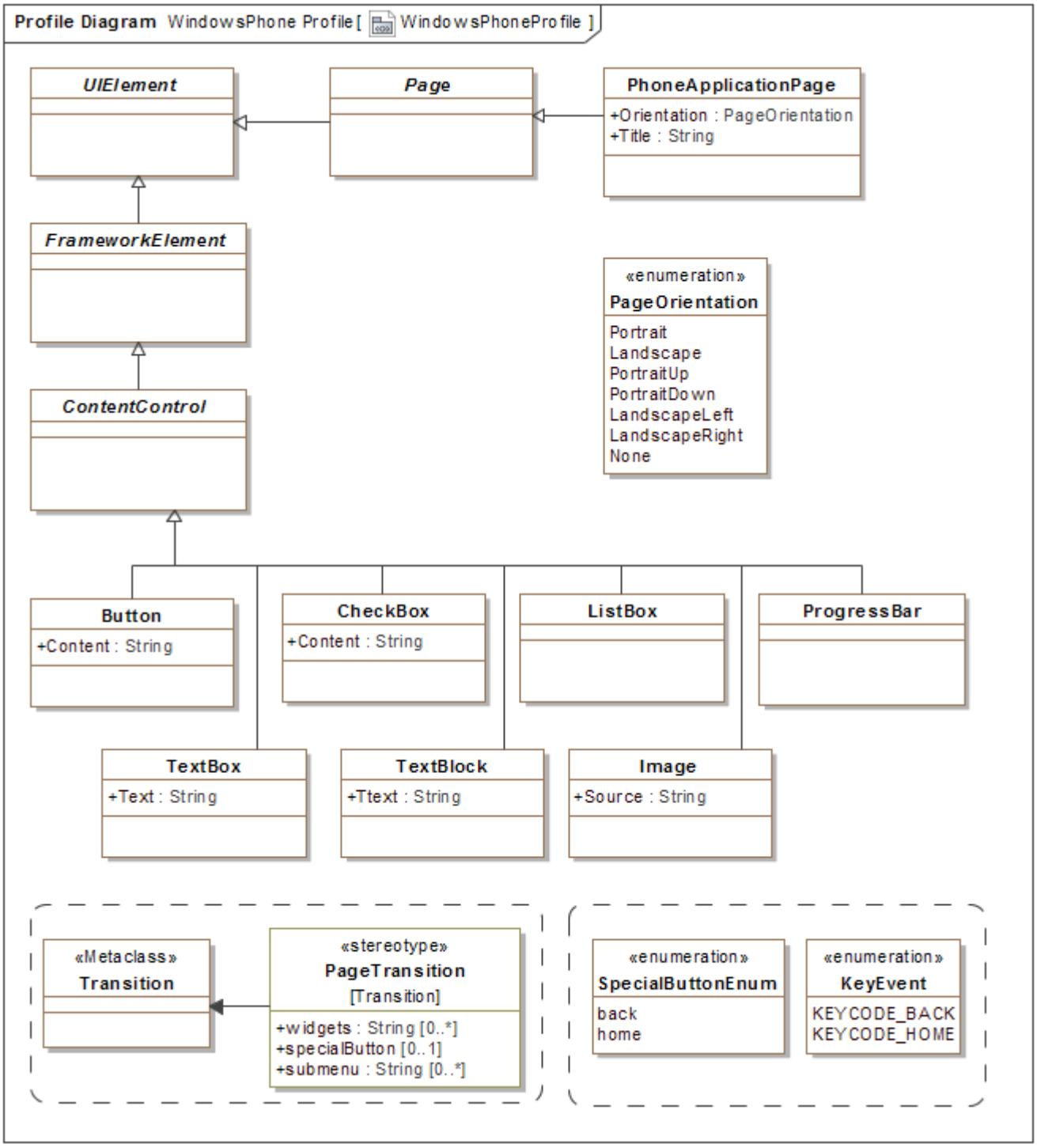


Figure 5.13: UML Profile for WindowsPhone7 PSM

work on UML metamodel elements (*Generalization association*, *Interface realization*, *Instance Specification*, *Dependency relationship* and so on), make use of syntactic constructs such as *For each* or *Exists* and tables mapping PIM elements into PSM elements.
To make more accurate, concise and understandable the rules it is useful to introduce some sets, definitions and notations:

- *Generalization association* from class A to class B is an UML association that links the specific class A to the generic class B. Namely, B generalizes the class A and A specializes the class B.

- *Interface realization* of interface I by class C is an UML specialized type of implementation relationship between a classifier and a provided interface.

- *Instance Specification* I of a class C, is an element that represents an instance of the class C from the Class Diagram. Instance Specification uses *slots* to show the attributes of the object. Each slot corresponds to a single attribute or feature, and may include a value for that entity.

- *Dependency relationship* from an Instance Specification I_1 to another Instance Specification I_2 is called, in UML, a supplier-client relationship, where supplier provides something to the client, and thus the client is in some sense incomplete while semantically or structurally dependent on the the supplier element

- $WIDGETS = \{w | w$ is a widget class of PIM $\}$. The set collects all widget classes of the PIM

- Dot notation $X.attr$ is used to refer to an instance specification *slot*, a class attribute or a XML tag attribute.

- The notation [...], inside a block of code, represent a parameterized field.

5.4 PIM to Android-PSM Transformation

The next sections describe in detail the rules implementing the transformation from PIM to Android Platform Specific Model. Rules 1-7, reported in section 5.4.1, perform the transformation of the Class Diagram, and generate the new Class Diagram, the android manifest file and the resources file `String.xml`. Rules 8-13, reported in section 5.4.2, perform the transformation of the Object Diagram, and generate, for each screen composing the application, the XML layout file. They also generate Android resource file for each menu of the Screens, and other specific layout files, such as the layout file for the ListView entries. Finally, the following rule performs the transformation of the UML Statechart Diagram:

Statechart Diagram Transformation

- Copy all *states*, *transitions* and the initial *pseudo-state* in the new PSM Statechart Diagram.
- For each *Transition* from state Q_1 to state Q_2, on which is applied the *ScreenTransition* stereotype, apply the *ActivityTransition* stereotype to the *Transition* counterpart element in the PSM Statechart Diagram.
- Map the content of the tagged values in the source *Stereotype* to the respective tagged values in target *Stereotype* of type `String`.

This mapping is necessary because in PIM the tagged values of stereotype *ScreenTransition* refer to *Instance Specification* elements, no longer present in PSM, because the Object Diagram has been transformed in XML Layout files.

With regard to the adaptation of the layout to different screen resolutions, Android transparently handles any scaling of the dp units, as necessary, based on the actual density of the screen in use. In this way it ensures proper display of your UI on screens with different densities. For this reason the coordinates of the widget can be copied from PIM to Android-PSM, without any conversion.

5.4.1 Structure Transformation

1. **Initialization**
   Create a new Class Diagram and apply to it the Android-PSM UML Profile.

2. **ProfileClass Mapping**
   This rule maps a PIM class in the Android specific one.

PIM	Android PSM
Widget	View
Screen	Activity
Label	TextView
Textbox	EditView
Button	Button
Checkbox	CheckBox
ImageBox	ImageView
ProgressBar	ProgressBar
Listbox	ListView

3. **ResourcePermissions Mapping**
   This rule maps a PIM resource permission in the Android specific one.

PIM	Android PSM
Camera	android.permission.CAMERA
ExternalStorage	android.permission.WRITE_EXTERNAL_STORAGE
PIM	android.permission.READ_CONTACTS
	android.permission.WRITE_CONTACTS
Network	android.permission.INTERNET

4. **Structure Mapping**

   - Add, to the Class Diagram, the classes `Activity`, `View` and the interfaces `OnClickListener` and `OnItemClickListener`.
   - For each *Generalization association* from class X to class `Screen`, add the class X to the target Class Diagram. Link class X, by the *Generalization association*, to the class `Activity`. Link class X, by *Interface realization* relationship, to the interface `OnClickListener`. Copy attributes and methods from the source classes, applying the rule **ProfileClass Mapping** to each attribute, method parameter and method return value.
   - For each *Generalization association* from class X to class W, where $W \in WIDGETS$, map, by **ProfileClass Mapping** rule, the class W to the Android specific one and add it to the target Class Diagram. Add the class X to the PSM and link it, by the *Generalization association*, to the Android counterpart class

of widget W. Finally, copy attributes and methods from the class X, applying the **ProfileClass Mapping** rule to each attribute, method parameter and method return value.

- For each class C, where $C \notin WIDGETS \cup \{$ Screen, Menu, Submenu, Application, AbstractDevice, ListboxItem $\}$, add a new class named C in PSM Class Diagram, and copy attributes and methods from the class C, applying the **ProfileClass Mapping** rule to each attribute, method parameter and method return value.
- For each *Interface realization* of interface I by class C, where $I \notin \{$ `onClickListener, onItemClickListener` $\}$ add a new interface I and link it, by *Interface realization*, to class C in PSM.

5. **Interfaces Implementation**
   For each *Instance Specification* I of class W, where $W = Listbox$, if exists a *Dependency relationship* from instance specification S to instance specification I, where S is an instance of a class P specializing `Screen`, then link the class P, by *Interface realization* relationship, to the interface `OnItemClickListener`.

6. **Resources File** Create the `String.xml` file, that will contain the string-resources of the application. The first string that the file must contain is the application name, retrieved from the `title` attribute[2] of the `Application` class.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">[application title]</string>
</resources>
```

7. **Manifest File**
   Let I the instance of the class `Application` in the Object Diagram. Create the file `AndroidManifest.xml`, starting with the following XML code:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="univr.[I.title]"
    android:versionCode="1"
    android:versionName="[I.version]" >
    <uses-sdk android:minSdkVersion="9"/>
```

   followed by the declaration of the resource permissions list, retrived from the `resources` attribute of the class `Application`, and translated by the rules **ResourcePermissions Mapping**

[2]Refer to the Instance Specification of the class in Object Diagram

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.INTERNET"/>
...
```

followed by the list of the Activity. For each *Generalization association* from class S to class `Screen`, add the following tag:

```
<activity android:name="[I.package].activity.[S]"/>
```

5.4.2 GUI Layout Transformation

8. **Layout Initialization**
   For each *Instance specification* I of class C, where C specializes `Screen` class, create an XML file with the same instance name I, and `.xml` suffix, starting with the following code:

```
<?xml version="1.0" encoding="utf-8"?>
```

9. **String Externalization**

   - Let T = {Label, Textbox, Button, Checkbox, Menu}
     For each *Instance Specification* I of class C, where $C \in T$, if exists a *Dependency relationship* from instance specification S to instance specification I, where S is an instance of the class`Screen`, then append to the `String.xml` file, created by **Resources File** rule, the tag:

```
<string name="[S]_[I]_text">[I.text]</string>
```

   - For each *Instance Specification* SM of class `Submenu`, if exists a *Dependency relationship* from instance specification M to instance specification SM, where M is an instance of the class`Menu`, then append to the file `String.xml`, created by the **Resources File** rule, the tag

```
<string name="[M]_[SM]_text">[SM.text]</string>
```

   - For each *Instance Specification* I of class `Listbox`, if exists a *Dependency relationship* from instance specification S to instance specification L, where S is an instance of the class`Screen`, then append to the file `String.xml`, created by **Resources File** rule, the following tag

```
<string-array name="[S]_[I]_items">
  ...
</string-array>
```

containing a tag `<item>` for each *Dependency relationship* from instance specification I to instance specification Q, where Q is an instance of the class `ListboxItem`

```
<item>[Q.text]</item>
```

10. **Screen Attributes Mapping**
For each *Instance specification* I of class S, where S specializes `Screen` class, map each slot reported in the following table to the corresponding Android XML attribute in the tag`<Activity>`, previously created in the file `AndroidManifest.xml` by the rules **Manifest File**:

Screen	**AndroidManifest.xml**
orientation	`android:screenOrientation` attribute of the tag `<activity>`
title	`android:label` attribute of the tag `<activity>`
fullscreen [true]	`android:theme` attribute of the tag `<activity>`
fullscreen [false]	nothing to map

```
<activity android:name="name"
        android:screenOrientation="[I.orientation]"/>
```

```
<activity android:name="name"
        android:label="[I.title]"/>
```

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
```

and append, to the XML layout file correspoding to the screen S, the Android-XML tag that maps the attribute `layout`:

layout slot	Screen Layout XML File
Linear	append the tag `<RelativeLayout>`

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"

</RelativeLayout>
```

Relative — append the tag `<LinearLayout>`

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

</LinearLayout>
```

11. **Screen Layout Mapping**
For each *Dependency relationship* from instance specification I_1 to instance specification I_2, where I_1 is an instance of a class S specializing `Screen` and I_2 is an instance of a class $W \in WIDGETS$, append to the XML layout file of the screen S the Android-XML tag that maps the widget W:

Widget	Android XML tag
Label	`<TextView android:id="@+id/[`I_2`]">`
Textbox	`<EditView android:id="@+id/[`I_2`]">`
Button	`<Button android:id="@+id/[`I_2`]">`
Checkbox	`<Checkbox android:id="@+id/[`I_2`]">`
Imagebox	`<ImageView android:id="@+id/[`I_2`]">`
Progressbar	`<Progressbar android:id="@+id/[`I_2`]">`
Listbox	`<Listview android:id="@+id/[`I_2`]">`

12. **Widget Attributes Mapping**
For each *Instance Specification* I of class W, where $W \in WIDGETS$, if exists a *Dependency relationship* from instance specification S to instance specification I, where S is an instance of a class P specializing `Screen`, then map the slots of instance specification I to the appropriate Android XML attributes, according to the following tables.

Attribute	Android XML	Value format
width	`android:layout_width`	
height	`android:layout_height`	

and if the slot `layout` of instance specification S is equal to `Relative`

Attribute	**Android XML**	**Value format**
above	`android:layout_above`	@+id/[I.above]dp
below	`android:layout_below`	@+id/[I.below]dp
toLeftOf	`android:layout_toLeftOf`	@+id/[I.toLeftOf]dp
toRightOf	`android:layout_toRightOf`	@+id/[I.toRightOf]dp
marginTop	`android:layout_marginTop`	[I.marginTop]dp
marginBottom	`android:layout_marginBottom`	[I.marginBottom]dp
marginLeft	`android:layout_marginLeft`	[I.marginLeft]dp
marginRight	`android:layout_marginRight`	[I.marginRight]dp

13. **Widget-Specific Attributes Mapping**
For each *Instance Specification* I of class W, where $W \in WIDGETS$, if exists a *Dependency relationship* from instance specification S to instance specification I, where S is an instance of a class P specializing `Screen`, then map the slots of instance specification I to the appropriate Android XML attributes, according to the following tables.
Mapping table for the widgets `Label, Textbox, Checkbox` and `Button`:

Attribute	**Android XML**	**Value format**
text	android:text	@string/[S]_[I]_text

Mapping table for `Progressbar` widget

Attribute	**Android XML**
maxValue	android:max

Mapping table for the `Listbox` widget:

Attribute	**Android XML**	**Value format**
items	android:entries	@array/[S]_[I]_items

Mapping table for the `Imagebox` widget:

Attribute	**Android XML**	**Value format**
imageSource	android:src	@drawable/[imageSource]

14. **Menu Mapping**

- For each *Instance Specification* S of class `Screen`, if exists a *Dependency relationship* from instance specification S to instance specification M, where M is an instance of the class `Menu`, then create an XML file named `[S]_menu.xml`, starting with the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/[S]_main_menu"
        android:title="@string/[S]_[M]_text">
    <menu> ... </menu>
    </item>
</menu>
```

- For each *Dependency relationship* from instance specification M to instance specification SM, where SM is an instance of the `Submenu` class, add the XML tag `<item>`.

```
<item android:id="@+id/[S]_[SM]"
    android:title="@string/[M]_[SM]_text"/>
```

5.5 PIM to WindowsPhone-PSM Transformation

The next sections describe in detail the rules implementing the transformation from PIM to Windows Phone 7 PSM. Rules 1-5, reported in section 5.5.1, perform the transformation of the Class Diagram, and generate the new Class Diagram and the manifest file of the application. Rules 6-11, reported in section 5.5.2, perform the transformation of the Object Diagram, and generate, for each screen composing the application, the corresponding XAML layout file. Finally, the following rule performs the transformation of the UML Statechart Diagram:

Statechart Diagram Transformation

- Copy all *states*, *transitions* and the initial *pseudo-state* in the new PSM Statechart Diagram.
- For each *Transition* from state Q_1 to state Q_2, on which is applied the *ScreenTransition* stereotype, apply the *PageTransition* stereotype to the *Transition* counterpart element in the PSM Statechart Diagram.
- Map the content of the tagged values in the source *Stereotype* to the respective tagged values in target *Stereotype* of type `String`.

This mapping is necessary because in PIM the tagged values of stereotype *ScreenTransition* refer to *Instance Specification* elements, no longer present in PSM, because the Object Diagram has been transformed in XAML Layout files.

As mentioned in section 3, the current Windows Phone 7 devices have the screen resolution of 480x800 pixels. The actual portion of the screen in which pages can be showed is smaller, 480x720 pixels. This resolution has the same aspect-ratio of that adopted in the PIM. Hence, in all the transformation rules that involve coordinates of the widgets, this coordinates must be scaled by a factor 1.5

5.5.1 Structure Transformation

1. **Initialization**
   This rule defines two global constants, $NAMESPACE$ and $MAINPAGE$.

- Create a new Class Diagram and apply to it the WindowsPhone-PSM UML Profile.
- Let I the *Instance specification* of the class `Application` in the PIM Object Diagram.
- Let T the unique outgoing *Transition* from initial pseudo state and state Q.
- Let `NAMESPACE` the value of the slot `title` of I
- Let `MAINPAGE` the name of state Q

2. **ProfileClass Mapping**
   This rule maps a PIM class in the Windows Phone specific one.

PIM	Windows Phone PSM
Widget	ContentControl
Screen	PhoneApplicationPage
Label	TextBlock
Textbox	TextBox
Button	Button
Checkbox	CheckBox
ImageBox	Image
ProgressBar	ProgressBar
Listbox	ListBox

3. **ResourcePermissions Mapping**
   This rule maps a PIM resource permission in the WindowsPhone specific one.

PIM	WP7 PSM
Camera	`ID_CAP_CAMERA`
	`ID_HW_FRONTCAMERA`
	`ID_CAP_ISV_CAMERA`
ExternalStorage	`ID_CAP_MEDIALIB`
PIM	`ID_CAP_APPOINTMENTS`
	`ID_CAP_CONTACTS`
Network	`ID_CAP_NETWORKING`
Phone	`ID_CAP_PHONEDIALER`

4. **Structure Mapping**

   - Add to the Class Diagram the classes `PhoneApplicationPage`, `ContentControl`.

- For each *Generalization association* from class X to class `Screen`, add the class X to the target Class Diagram. Link class X, by the *Generalization association*, to the `PhoneApplicationPage` class. Copy also attributes and methods from the source classes, applying the **ProfileClass Mapping** rule to each attribute, method parameter and method return value.
- For each *Generalization association* from class X to class W, where $W \in WIDGETS$, map, by **ProfileClass Mapping** rule, the class W to the WindowsPhone specific one and add it to the target Class Diagram. Add the class X to the PSM and link it, by the *Generalization association*, to the WindowsPhone counterpart class of widget W. Finally, copy attributes and methods from the class X, applying the **ProfileClass Mapping** rule to each attribute, method parameter and method return value.
- For each class C, where $C \notin WIDGETS \cup \{$ Screen, Menu, Submenu, Application, AbstractDevice, ListboxItem $\}$, add a new class named C in PSM Class Diagram, and copy attributes and methods from the class C, applying the **ProfileClass Mapping** rule to each attribute, method parameter and method return value.
- For each *Interface realization* of interface I by class C, add a new interface I and link it, by *Interface realization*, to class C in PSM.

5. **Manifest File**
   Let I the *Instance specification* of the class `Application` in the PIM Object Diagram. Create the `WMAppManifest.xml` file, starting with the following XML code

```
<?xml version="1.0" encoding="utf-8"?>
<Deployment
  xmlns="http://schemas.microsoft.com/windowsphone/2009/deployment"
  AppPlatformVersion="7.0">
  <App xmlns="" Title="[I.title]" RuntimeType="Silverlight"
       Version="[I.version]" Genre="apps.normal"
       Author="" Description="" Publisher="">
    <IconPath IsRelative="true" IsResource="false">ApplicationIcon.png</IconPath
        >
```

   followed by the declaration of the resource permissions list, retrived from the `resources` attribute of the class `Application`, and translated by the rule **ResourcePermissions Mapping**

```
<Capabilities>
  <Capability Name="ID_CAP_NETWORKING"/>
  <Capability Name="ID_CAP_PHONEDIALER"/>
  ...
</Capabilities>
```

followed by

```
<Tasks>
  <DefaultTask Name ="_default" NavigationPage="MainPage.xaml"/>
</Tasks>
<Tokens>
  <PrimaryToken TokenID="MapNavigatorToken" TaskName="_default">
    <TemplateType5>
      <BackgroundImageURI IsRelative="true"
                  IsResource="false">
        Background.png
      </BackgroundImageURI>
      <Count>0</Count>
      <Title>MapNavigator</Title>
    </TemplateType5>
  </PrimaryToken>
</Tokens>
</App></Deployment>
```

The tag `DefaultTask` will be setted to the appropriate value by the **Statechart** rule.

5.5.2 GUI Layout Transformation

6. **Layout Initialization**
   For each *Instance specification* I of class C, where C specializes `Screen` class, create an XML file with the same instance name I, and `.xaml` suffix, starting with the following code:

```
<phone:PhoneApplicationPage
    x:Class="[NAMESPACE].[C]"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="" Orientation=""
    shell:SystemTray.IsVisible="True"
    Loaded="PhoneApplicationPage_Loaded">

    <Grid x:Name="LayoutRoot" Background="Transparent" ShowGridLines="False">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,0">
            <TextBlock x:Name="PageTitle" Text="[I.title]" Style="{StaticResource
                PhoneTextNormalStyle}"/>
        </StackPanel>

      <Grid x:Name="ContentPanel" Grid.Row="1" Margin="0,0,0,0">
        </Grid>
```

```
29      </Grid>
30  </phone:PhoneApplicationPage>
```

7. **Screen Attributes Mapping**
   For each *Instance specification* I of class S, where S specializes `Screen` class, map each slot reported in the following table to the corresponding XAML attribute of the tag `<phone:PhoneApplicationPage>` in S`.xmal` file, previously created by the **Layout Initialization** rule:

Screen	XAML attribute
orientation	`SupportedOrientations` `Orientation`
fullscreen [true]	`shell:SystemTray.IsVisible="false"` and remove the tag `StackPanel x:Name="TitlePanel"` from xaml file
fullscreen [false]	`shell:SystemTray.IsVisible="true"`

   The attribute `title` is mapped into the attribute `Text` of the tag ¡StackPanel x:Name=”TitlePanel”¿

```
1  <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,0">
2    <TextBlock x:Name="PageTitle" Text="[I.title]" Style="{StaticResource␣
         PhoneTextNormalStyle}"/>
3  </StackPanel>
```

8. **Screen Layout Mapping**
   For each *Dependency relationship* from instance specification I_1 to instance specification I_2, where I_1 is an instance of a class S specializing `Screen` and I_2 is an instance of a class $W \in WIDGETS$, append to the file S`.xaml`, within the tag `<Grid x:Name="ContentPanel">`, the XAML tag that maps the widget W:

Widget	XAML tag
Label	`TextBlock Name="[`I_2`]"`
Textbox	`TextBox Name="[`I_2`]"`
Button	`Button Name="[`I_2`]" Click="[`I_2`]_Click"`
Checkbox	`CheckBox Name="[`I_2`]" Click="[`I_2`]_Click"`
Imagebox	`Image Name="[`I_2`]"`
Progressbar	`ProgressBar Name="[`I_2`]"`
Listbox	`ListBox Name="[`I_2`]" SelectionChanged="[`I_2`]_SelectionChanged"`

9. **Widget Attributes Mapping**
For each *Instance Specification* I of class W, where $W \in WIDGETS$, if exists a *Dependency relationship* from instance specification S to instance specification I, where S is an instance of a class P specializing `Screen`, then map the slots of instance specification I to the appropriate XAML tag attributes, according to the following tables.

PIM Attribute	XAML tag attribute
width	`Width`
height	`Height`

In Windows Phone, the XAML layout description adopts the *absolute layout* in which each widget specifies the distances from the top-left corner of the screen. The two recursive functions *getX* and *getY*, introduced in section 5.1.3, can be used to map the PIM positioning attributes to the PSM specific one.

XAML attribute	Value
Margin	"[getX(W)],[getY(W)],0,0"

10. **Widget-Specific Attributes Mapping**
For each *Instance Specification* I of class W, where $W \in WIDGETS$, if exists a *Dependency relationship* from instance specification S to instance specification I, where S is an instance of a class P specializing `Screen`, then map the slots of instance specification I to the appropriate XAML attributes, according to the following table:

Widget	Attribute	XAML attribute
Label	text	Text
Textbox	text	Text
Button	text	Content
Checkbox	text	Content
Progressbar	maxValue	Maximum
Imagebox	imageSource	Source

11. **ListView Items Mapping**
For each *Instance Specification* I of class `Listbox`, if exists a *Dependency relationship* from instance specification S to instance specification I, where S is an instance of a class P specializing `Screen`, then for each *Dependency relationship* from instance specification I to instance specification Q, where Q is an instance of the class `ListboxItem`, add the following nested tag

```
<ListBoxItem Content="[Q.text]" />
```

in the corrispondent tag `<ListBox Name="[I]">`

12. **Menu Mapping**

- For each *Instance Specification* S of class `Screen`, if exists a *Dependency relationship* from instance specification S to instance specification M, where M is an instance of the `Menu` class, then append to the file S`.xaml`, within the tag `<phone:PhoneApplicationPage>`, the following XAML code;

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBar.MenuItems>

        </shell:ApplicationBar.MenuItems>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

- For each *Dependency relationship* from instance specification M to instance specification SM, where SM is an instance of the `Submenu` class, add within the tag `<shell:ApplicationBar.MenuItems>`, the following code:

```
<shell:ApplicationBarMenuItem Text="[SM.text]" Click="[SM]_Click"/>
```

5.6 Code Generation

The final step of the Model-Driven Design is the generation of the application code from each Platform Specific Model. The generated code is both structural and behavioral code. The structural code concerns the structure of class files, that is constructors, interfaces implementations and methods signatures. The behavioral code implements some specific operations, like screens transitions, *Back button* redirection or the initialization of GUI. So, the code generation procedure is based on information coming from the PSM Class Diagram and from the UML Statechart Diagram for the behavioral aspects. This procedure can be implemented by defining a number of rules that involve a set of parameterized blocks of code.
To make more accurate, concise and understandable the rules it is useful to introduce some sets, definitions and notations:

- *State* Q representing the Activity A
- $WIDGETS = \{w | w$ is a widget class of AndroidPSM $\}$. The set collects all widget classes of the PSM
- Dot notation $X.attr$ is used to refer to an instance specification *slot*, a class attribute or a XML tag attribute.
- The notation [...], inside a block of code, represent a parameterized field.

To obtain a complete Eclipse[3] project or VisualStudio[4] Project, it is necessary to create some other xml and plain-text files, not strictly depending from the particular application that we are modeling. The discussion of these files will not be detailed.
The next two sections introduce separately the set of code-generation rules for both Android and WindowsPhone platform.

[3]Eclipse is an open-source community that develops open platforms and products. Google provides the Android Development Tools (ADT), a plugin for the Eclipse IDE that is designed to give you a powerful, integrated environment in which to build Android applications.

[4]Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop console and graphical user interface applications along with Windows Forms applications, web applications, web services and smartphone application in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Phone, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

5.7 Android Code Generation

Android associates a constant to each resource of the application, such as Strings, Layout, Menu, View and Images. These constants make it possibile to refer to the resources within the application code. So, the first step of code generation is to create the resource file `R.java`. Rules 3-4 generate the structural part of the code and rules 7-9 introduce the blocks of code that manage the activity transitions.

1. **Resource File Initialization**

   - Create the `R.java` class file starting with the following code:

```
public final class R {
  public static final class string {
    public static final int app_name=0x7f1;
  }
  public static final class array { }
  public static final class id { }
  public static final class layout { }
  public static final class drawable { }
}
```

   - Let C an integer variable, initialized to 1, to the resource file `R.java`, by the current rule. This variable will be incremented by one every time a new resource constant will be added to the resource file.
   - For each `<String>` tag T in `String.xml` file add, to the nested class `String` in class `R.java`, the following line of code:

```
public static final int [T.name]=0x7f[C];
```

   - For each `<string-array>` tag T in `String.xml` file add, to the nested class `array` in class `R.java`, the following line of code:

```
public static final int [T.name]=0x7f[C];
```

   - For each XML Layout file L (generated by **Layout Initialization** rule introduced in section 5.4.2), refering to the Activity A, add to the nested class `layout` in class `R.java` the following line of code:

```
public static final int layout_[A]=0x7f[C];
```

   - For each XML file L (generated by **Menu Mapping** rule introduced in section 5.4.2), refering to the Menu M, add to the nested class `layout` in class `R.java`, the following line of code:

```
public static final int [M]=0x7f[C];
```

- For each `drawable` resource D (generated by **Widget-Specific Attributes Mapping** introduced in section 5.4.2), add to the nested class `drawable` in class `R.java`, the following line of code

```
1 public static final int drawable_[D]=0x7f[C];
```

- For each XML Layout file L (generated by **Layout Initialization** rule introduced in section 5.4.2), and for each XML tag T, where $T \in WIDGETS$, add to the nested class `id` in class `R.java`, the following line of code:

```
1 public static final int [T.android:id]=0x7f[C];
```

2. **Package Generation**

   - Let T the `<manifest>` tag of `AndroidManifest.xml` file (generated by the **Manifest File** rule introduced in section5.4.1).
   - Create a Java package named [T`.package`].
   - Create three other packages:
     [`T.package`].activity, [`T.package`].widgets and [`T.package`].util

3. **Structural Code Generation**
   The generation of structural code can be done by almost all UML Modeling tools, and it simply transforms, in Java code, the structure of the classes from UML Class Diagram. The procedure generates java code from *classes, attributes, operation signatures, interface realizations* and *generalization associations*, appending all this code to new java files. So, the output of this step is a set of raw java files (one for each class in the PSM Class Diagram) that must be completed with import directives and several blocks of code.

4. **Import Directives Generation**
   Let T the `<manifest>` tag of `AndroidManifest.xml` file (generated by the **Manifest File** rule). For each *Generalization association* from class A to class `Activity`, insert in the file [A].`java` the following `import` directives:

```
1  package [T.package].activity;
2  import [T.package].R;
3  import android.app.Activity;
4  import android.content.Context;
5  import android.content.Intent;
6  import android.view.View;
7  import android.view.View.OnClickListener;
8  import android.widget.AdapterView.OnItemClickListener;
9  import android.os.Bundle;
10 import android.widget.*;
```

For each *Generalization association* from class C to class W, where $W \in WIDGETS$, insert in the file [C].`java` the following `import directives`:

```
1 package [T.package].widgets;
2 import [T.package].R;
3 import android.content.Context;
4 import android.view.View;
5 import android.widget.[W];
```

5. **Widgets Initialization**
   For each Activity layout file F, and for each XML tag T in F, where $T \in WIDGETS$, add to the class F`.java` the private class variable

```
1 private [T] [T.android:id];
```

   and the follow initialization code in the overridden method `onCreate()`

```
1 [T.android:id] = ([T]) findViewById(R.id.[T.android:id]);
2 if ([T.android:id] != null) [T.android:id].setOnClickListener(this);
```

   For each Activity layout file F, and for each XML tag `<ListView>`, add the follow initialization code in the overridden method `onCreate()`

```
1 if ([T.android:id] != null) [T.android:id].setOnItemClickListener(this);
```

6. **GUI Layout Initialization**
   For each Activity A:

   - Let M the `<manifest>` tag of the corresponding XML Layout file
   - insert in the overridden method `public void onCreate(Bundle savedInstanceState)` of the class A, the following lines of code:

```
1 super.onCreate(savedInstanceState);
2 setContentView(R.layout.[A]);
3 setTitle([M.android:label]);
```

7. **Activity Menu Initialization**
   For each Activity A, if exits the XML file A`_menu.xml` (possibly created by **Menu Mapping** rule introduced in section 5.4.2), insert in the overridden method `onCreateOptionsMenu()` of class A, the following lines of code:

```
1 MenuInflater inflater = getMenuInflater();
2 inflater.inflate(R.menu.[A]_main_menu, menu);
3 return true;
```

8. **Widget Methods Initialization**

   - For each Activity layout file F, insert, in the method `onClick` of class file F`.java` in the `.activity` package the `switch` statement:

```
switch (v.getId()) { }
```

For each XML tag T in F, where $T \in \{$ `Button`, `CheckBox` $\}$, add a `case` statement

```
public void onClick(View v) {
  switch (v.getId()) {
    case R.id.[T]:
      [T]_Click();
      break;
    case ...:
      ...
      break;
  }
}
```

and add the private class method `[T]_Click()`

- For each Activity layout file F insert, in the method `onItemClick` (if exists) of class file F`.java` in the `.activity` package, the `switch` statement and variable declaration:

```
Intent intent;
switch (arg0.getId()) { }
```

For each XML tag T of type `ListView` in F, add a `case` statement

```
public void onItemClick(AdapterView<?> arg0, View arg1, int arg2, long arg3) {
  switch (arg0.getId()) {
    case R.id.[T]:
      [T]_ItemClick(arg2);
      break;
    case ...:
      ...
      break;
  }
}
```

and add the private class method `[T]_ItemClick(int item)`

9. **Menu Methods Initialization** For each Activity A, if exits the XML file A`_menu.xml` (possibly created by **Menu Mapping** rule introduced in section 5.4.2) then:

   - insert in the overridden method `onOptionsItemSelected` of class file A`.java`, in the `.activity` package, the `switch` statement:

```
switch (item.getItemId()) { }
```

   - For each `<item>` tag I in the file XML file A`_menu.xml`, add a `case` statement to the `Switch` in the method `onOptionsItemSelected`

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.[I.androi:id]:
      [I.androi:id]_Click();
```

```
6         break;
7       case ...:
8         ...
9         break;
10    }
11  }
```

10. **Transitions by Widget Interaction**

    - For each *Transition* from state Q to state T, for each entry W of the `wigets` tagged value of stereotype applied to the transition, add to the private class method `[W]_Click()` the following code:

```
1  Intent intent = new Intent(this, [T].class);
2  startActivity(intent);
```

    - For each *Transition* from state Q to state T, for each entry W of the `wigets` tagged value of stereotype applied to the transition, where W = `ListView`, add to the private class method `[W]_ItemClick()` the following code:

```
1  intent = new Intent(this, [T].class);
2  startActivity(intent);
```

11. **Transitions by Menu Interaction**
    For each *Transition* T from state Q to state P and for each entry SM of the tagged value `submenu` of the stereotype applied to T, add to the class method `[SM]_Click` the following code:

```
1  Intent intent = new Intent(this, [P].class);
2  startActivity(intent);
```

12. **Transitions by Method Invocation**
    For each *State* Q and for each outgoing *Transition* T from state Q to state S insert, to the method specified by the attribute *operation* of T, the following code:

```
1  Intent intent = new Intent(this, [S].class);
2  startActivity(intent);
```

13. **Transitions by Back-Button Redirection**
    For each *State* Q, if exists an outgoing *Transition* T from state Q to state P, for which the tagged value `specialButton` is defined, then add to the overridden method `onKeyDown` of the class Q the following code:

```
1  public boolean onKeyDown(int keyCode, KeyEvent event) {
2    switch(keyCode) {
3      case KeyEvent.KEYCODE_BACK:
4        Intent intent = new Intent(this, [P].class);
```

```
5         startActivity(intent);
6         break;
7     }
8     return super.onKeyDown(keyCode, event);
9 }
```

14. **Guards on Transitions**
    Whenever on a transition is specified a guard (in term of OCL boolean expression), an `if ( .. )` statement must be placed before the transition code. Translating OCL expressions in Java code is a complex task, but for a subset of the OCL language (in particular, boolean expression) this operation can be simply automated by external tools, like the Dresden OCL Library[5], obtaining compact java Code. For a more detailed discussion refer to *Compilation of OCL into Java for the Eclipse OCL Implementation* [9].

5.8 Windows Phone Code Generation

The code generation for Windows Phone platform is a bit different. Usually the generation of structural code can be done by almost all UML Modeling Tools, and it simply transforms, in target code, the structure of the classes from UML Class Diagram. The generation of CSharp classes is a more complex task, because in the VisualStudio project for Windows Phone, some classes are separated in two partial classes, one of which contains code that is auto-generated (and hidden to the developer) by the IDE. This holds only for those classes ,specializing PhoneApplicationPage, that contain the code of the application *Pages*. Code Generation rules must generate files and classes conform to those (partially) auto-generated by the VisualStudio IDE.

1. **Initialization**

   - Let I the *Instance specification* of the class `Application` in the PIM Object Diagram.
   - Let `NAMESPACE` the value of the slot `title` of I

   This rule defines the global constants $NAMESPACE$.

2. **Structural Code Generation**
   For each *Generalization association* from class X to class `PhoneApplicationPage`, create a first file named X`.xaml.cs` starting with the following code:

[5] http://www.dresden-ocl.org/index.php/DresdenOCL

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls.Primitives;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.Phone.Controls;
using System.Windows.Threading;

namespace [NAMESPACE] {
    public partial class [X] : PhoneApplicationPage {
        public [X]() {
            InitializeComponent();
        }

        private void PhoneApplicationPage_Loaded(object sender, RoutedEventArgs e)
        {
          // custom initialization code
        }
    }
}
```

and a second file named X`.g.cs` with the following code:

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls.Primitives;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.Phone.Controls;
using System.Windows.Threading;

namespace [NAMESPACE] {
  public partial class MainPage :
Microsoft.Phone.Controls.PhoneApplicationPage {
        internal System.Windows.Controls.Grid LayoutRoot;
        internal System.Windows.Controls.StackPanel TitlePanel;
        internal System.Windows.Controls.TextBlock ApplicationTitle;
        internal System.Windows.Controls.Grid ContentPanel;
        private bool _contentLoaded;
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent() {
            if (_contentLoaded) {return; }
              _contentLoaded = true;
            System.Windows.Application.LoadComponent(this,
        new System.Uri("/[NAMESPACE];component/[X].xaml",
    System.UriKind.Relative));
            this.LayoutRoot =
    ((System.Windows.Controls.Grid)(this.FindName("LayoutRoot")));
            this.TitlePanel =
    ((System.Windows.Controls.StackPanel)(this.FindName("TitlePanel")));
              this.ApplicationTitle =
    ((System.Windows.Controls.TextBlock)(this.FindName("ApplicationTitle")));
```

```
35              this.ContentPanel =
36      ((System.Windows.Controls.Grid)(this.FindName("ContentPanel")));
37          }
38      }
39  }
```

3. **Widgets Initialization**
   For each XAML layout file F, and for each XAML tag T within the tag `<Grid x:Name="ContentPanel">` of F, where $T \in WIDGETS$, add to the partial class file F`.g.cs` the *internal* class variable

```
1  internal System.Windows.Controls[T] [T.Name];
```

   and the follow initialization code in the method `InitializeComponent`

```
1  this.[T] = ((System.Windows.Controls.[T])(this.FindName("[T.Name]")));
```

4. **Menu Methods Initialization**
   For each XAML layout file X, generated by the transformation rule **Layout Initialization**, for each XAML tag T of type `<shell:ApplicationBarMenuItem>` within the container tag `<phone:PhoneApplicationPage.ApplicationBar>` (if it exists), add to the class file X`.cs` the method

```
1  private void [T.Click](object sender, EventArgs e) { }
```

5. **Widget Methods Initialization**
   For each XAML layout file X, generated by the transformation rule **Layout Initialization**, for each XAML tag T within the container tag `<Grid x:Name="ContentPanel">`, add one event-handler private method according to the following table:

Widget	Event Handler
Button	`1` [T.Name]_Click(**object** sender, RoutedEventArgs e)
Checkbox	`1` [T.Name]_Click(**object** sender, RoutedEventArgs e)
Listbox	`1` [T.Name]_SelectionChanged(**object** sender, SelectionChangedEventArgs e)

6. **Transitions by Widget Interaction**
   For each *Transition* T from state Q to state S, for each entry W in the tagged value `wigets` of stereotype applied to the transition, where $W \in \{$ Button, CheckBox $\}$, add to the method W`_Click` of the partial class Q`.xaml.cs` the following code:

```
1  NavigationService.Navigate(new Uri("/[S].xaml", UriKind.Relative));
```

For each *Transition* T from state Q to state S, for each entry W in the tagged value `wigets` of stereotype applied to the transition, where $W =$ `ListBox`, add to the method W`_SelectionChanged` of the partial class Q`.xaml.cs` the following code:

```
1 NavigationService.Navigate(new Uri("/[S].xaml", UriKind.Relative));
```

7. **Transitions by Menu Interaction**
   For each *Transition* T from state Q to state S, for each entry SM in the tagged value `submenu` of stereotype applied to the transition, add to the method SM`_Click` of the partial class Q`.xaml.cs` the following code:

```
1 NavigationService.Navigate(new Uri("/[S].xaml", UriKind.Relative));
```

8. **Transitions by Method Invocation**
   For each *Transition* T from state Q to state S insert, to the method (of the class Q`.xaml.cs`) specified by the attribute *operation* of T, the following code:

```
1 NavigationService.Navigate(new Uri("/[S].xaml", UriKind.Relative));
```

9. **Transitions by Back-Button Redirection**
   For each *State* Q, if exists an outgoing *Transition* T from state Q to state P, for which the tagged value `specialButton` is defined, then add to the overridden method `OnBackKeyPress` of the class Q the following code:

```
1 protected override void OnBackKeyPress(CancelEventArgs e) {
2   NavigationService.Navigate(new Uri("/[P].xaml", UriKind.Relative));
3 }
```

10. **Guards on Transitions**
    Whenever on a transition is specified a guard (in term of OCL boolean expression), an `if ( .. )` statement must be placed before the transition code. Translating OCL expressions in CSharp code is a complex task, but for a subset of the OCL language (in particular, boolean expression) this operation can be automated by external tools. For a more detailed discussion refer to *Implementing an OCL Compiler for.NET* [10].

5.9 Computational Analysis

The transformation and code-generation rules are the core of the Model-Driven Design, and it is useful examining the computational complexity of

their implementation. On the one hand, the complexity can be expressed by evaluating, at the highest level, the number of elements that make up the application to be modeled. On the other hand, the elements of the metamodel can be taken as a reference at the lowest level. Transformations act linearly on the high-level elements, beacuse each element of the PIM is transformed into the corresponding element in PSM. Indeed, each target-platform version of the modeled application is still composed my the same structural elements introduced at platform-independent level. For example, **Structure Mapping** rule, the main transformation rule of the class diagram, translates each Screen and custom Widget modeled by developer, into the specific one for each target platform. **Layout Initialization** rule creates a XML layout file for each `Screen`, and **Widget Attributes Mapping** enriches the layout file setting the attributes of each Widget composing the Screen. Some other rules, such as **ProfileClass Mapping** and **ResourcePermission Mapping** (they are rules, like the previous, introduced both for Android and Windows Phone), work in constant time. At metamodel level, the number of elements increases linearly as the number of screens and widgets introduced in the PIM. Rules expressed in the form *For each Instance Specification I..., if exists Dependency Relationship D...*, such as **Widget Attribute Mapping**, **Menu Mapping** and several other, have a quadratic time complexity in the worst-case. Indeed, *Instance Specifications* set and *Dependency Relationships* set may have the same cardinality. The application in sequence of different rules introduces a multiplicative constant up to 8.
Despite these considerations, the complexity of the transformation and code-generation process is usually negligible, even in cases of applications with a large number of screens and widgets. Since the transformation rules are not implemented in a concrete transformation language, it is not possible to detail the analysis, which however can be used as base to improve the rules in a specific implementation.

Chapter 6

Experimental Validation

This chapter presents the Platform Independent Model of a sample application. The transformation and code-generation rules will be applied, to compare the resulting application structure with a version of the same application, previously developed for both Android and Windows Phone platform. The output of the rules will be shown by applying them manually, because the transformations and the code generation have not been implemented in a tool. The goal is to compare the code (XML GUI Layout and Java/CSharp code) obtained from an application designed at the Platform Independent Level, with the code and the structure of the same application, but developed in the traditional way for different platforms. To make the chapter more readable, the figures representing the various diagrams of PIM and PSM are moved to Appendix B. The same thing applies to the code listings, moved to Appendix C.

6.1 Application Example

The sample application is a restricted version of MapNavigator, an application developed during a work finished before the beginning of this thesis. MapNavigator was designed as a guide tool for the visit of fairs, exhibitions and museums. Briefly, MapNavigator is useful for organizers / exhibitors, when they have a map, a set of information about events or points of interest, and they want to define thematic routes to guide the visitors. For each point of interest (POI) a Quick Response Code[1] (QR Code), encoding

[1]QR Code is the trademark for a type of matrix barcode, first designed for the automotive industry. More recently, the system has become popular outside of the industry due to its fast readability and large storage capacity compared to standard UPC barcodes.

some static information about the POI, may be generated and placed where visitors, with their smartphone integrated camera, can acquire it in order to obtain information about the location where they are. For the exhibitors is available a tool, not shown in this discussion, that allows to configure the map and the related points of interest, and make it available to visitors, along with the application itself.

The application is structured according to the main activities that the user can perform:

- choose a map, from the internal or external storage memory (screen ChooseMap)
- consult the map and its points of interest (screen NavigateMap)
- acquire a QRCode with the integrated camera (screen AcquireQRCode)
- consult the points of interest list (screen TagList)
- read the information about a single point of interest (screen TagInfo)
- edit some basic preferences about the application (screen Preferences)

A QR Code can be acquired without having previously loaded a map. In this way the visitor can however obtain information about a point of interest, even if it cannot see the position on the map.

6.1.1 The Platform Independent Model

Figures B.1 to B.4 in Appendix B depict the diagrams composing the PIM. The Class Diagram in Figure 6.1 shows the seven classes, one for each Screen of the application, specializing the class `Screen`.

The GUI Layout of each screen is described by the Object Diagram reported in Figure B.1, B.2 and B.3. Finally, the Statechart Diagram describing the behavioural aspects of the application is depicted in Figure B.4

The code consists of block modules arranged in a square pattern on a white background. The information encoded ca be made up of four standardized kinds of data (numeric, alphanumeric, byte/binary, Kanji), or by supported extensions virtually any kind of data. (Source http://en.wikipedia.org/wiki/QR_code)

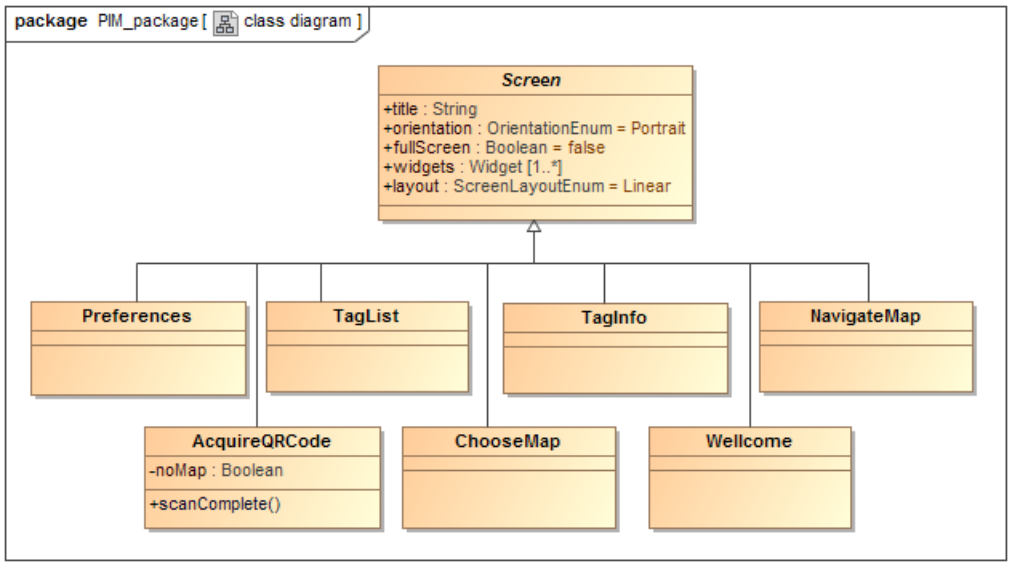


Figure 6.1: PIM Class Diagram

6.2 Evaluation Metrics

To compare the resulting application, obtained applying the transformations introduced in the previous chapter, it is necessary to define which aspects to consider. First, the comparison may be done considering the structural aspects and some behavioral aspects. It does not consider the specific implemented functionalities, since they are outside the scope of the proposed methodology.
The comparison can be made mainly by assessing the following aspects:

- the number of lines of code and the number of methods composing the classes and the GUI layout descriptions

- limitations due to the limited number of widgets provided by the UML Profile for the Platform Independent Model

These aspects regard the possible differences between the code obtained from the Model-Driven approach and from the traditional one. Furthermore, we may also compare how much similar are the User Interfaces of the applications obtained for each target platform.

6.3 Comparison with Traditional Implementation

This section shows the Platform Specific Model for the two platforms under examination, and in parallel compares the code obtained from the transformation and code-generation rules with the same application code previously manually developed.

6.3.1 Android

Rule 4 (Structure Mapping): Transformation rule that generates the class diagram depicted in Figure B.5. The Class Diagram shows the seven Screen classes that in Android must specialize the class `Activity`, and the interfaces implemented by some of them.

Rule 6 (Resource File) and Rule 9 (String Externalization): transformation rules which generate the file containing the strings, used by the widgets and externalized from the code. The following two listings show, in order, the file obainted by the transformation and the manually written:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="app_name">MapNavigator</string>
<string name="wellcome_buttonOpen_text">Open Map</string>
<string name="wellcome_buttonLast_text">Latest Map</string>
<string name="wellcome_buttonQRCode_text">Acquire QRCode</string>
<string name="wellcome_buttonPrefs_text">Preferences</string>
<string name="preferences_labelTitle_text">Preferences</string>
<string name="preferences_labelLocation_text">Default maps location</string>
<string name="preferences_labelLast_text">Save the latest map</string>
<string name="preferences_labelUpdate_text">Search for updates</string>
<string name="preferences_buttonSave_text">Save</string>
<string name="navigatemap_navmenu_text">Map</string>
<string name="navmenu_taglistMenu_text">Point of Interest list</string>
<string name="navmenu_qrcodeMenu_text">Acquire QRCode</string>
<string name="tagInfo_buttonMap_text">Go to map</string>
</resources>
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="app_name">MapNavigator</string>
<string name="label1">Open Map</string>
<string name="label2">Latest Map</string>
<string name="label3">Acquire QRCode</string>
<string name="label4">Preferences</string>
<string name="labelPrefs">Preferences</string>
<string name="labelPrefsLocation">Default map location</string>
<string name="labelPrefsLastmap">Save the latest map</string>
<string name="labelPrefsUpdate">Search for updates</string>
<string name="buttonSave">Save</string>
```

```
<string name="menuLabelMain">Map</string>
<string name="menuLabelTagList">Point of Interest list</string>
<string name="menuLabelQRCode">Acquire QRCode</string>
<string name="buttonMap">Go to Map</string>
</resources>
```

As can be seen, the main difference is the standardized format of the entries (`[activityName]_[widgetName]_text`). In the manually version the choice is up to the developer, and it may not follow a standard schema.

AndroidManifest.xml: the manifest file of the application, generated by the transformation rule **Manifest File**, together with the rule **Screen Attributes Mapping**. The code is the same of the manually developed, except for the name of some Activities.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="univr.MapNavigator"
    android:versionCode="1"
    android:versionName="1.0" >
      <uses-sdk android:minSdkVersion="8" />
  <uses-permission android:name="android.permission.CAMERA" />
  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <application
        android:icon="@drawable/ic_launcher" android:label="@string/app_name" >
    <activity android:name="univr.mapnavigator.activity.Wellcome"
        android:label="@string/app_name" android:screenOrientation="portrait">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
  <activity android:name="univr.mapnavigator.activity.ChooseMap"
      android:label="Choose_Map" android:screenOrientation="portrait"/>
  <activity android:name="univr.mapnavigator.activity.AcquireQRCode"
      android:label="Acquire_QR_Code" android:screenOrientation="landscape"/>
  <activity android:name="univr.mapnavigator.activity.NavigateMap"
      android:label="Map" android:screenOrientation="landscape"
      android:theme="@android:style/Theme.NoTitleBar.Fullscreen" />
  <activity android:name="univr.mapnavigator.activity.Preferences"
      android:label="Preferences" android:screenOrientation="portrait"/>
  <activity android:name="univr.mapnavigator.activity.TagList"
       android:label="Tag_List" android:screenOrientation="portrait"/>
  <activity android:name="univr.mapnavigator.activity.TagInfo"
       android:label="Tag_Info" android:screenOrientation="portrait"/>
  </application>
</manifest>
```

In the first part of the manifest there are the `<uses-permission>` tags that map the attribute `resources` of the PIM class `Application`, by the transformation rule **ResourcePermissions Mapping**.

GUI Layout The following parallel listings (divided into two pages for reasons of space) show, on the left, the XML file obtained by the transformation rules and, on the right, the same XML file manually designed. These files describe the GUI Layout of the `Wellcome` screen.

Auto-generated

```
<?xml version="1.0" encoding="utf−8"?>
<RelativeLayout xmlns:android="..."
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

<ImageView
android:id="@+id/imageOpen"
android:layout_width="72dp"
android:layout_height="72dp"
android:layout_marginLeft="40dp"
android:layout_marginTop="40dp"
android:src="@drawable/icon_open" />

<ImageView
android:id="@+id/imageLast"
android:layout_width="72dp"
android:layout_height="72dp"
android:layout_below="@+id/imageOpen"
android:layout_marginLeft="40dp"
android:layout_marginTop="20dp"
android:src="@drawable/icon_recent" />

<ImageView
android:id="@+id/imageQRCode"
android:layout_width="72dp"
android:layout_height="72dp"
android:layout_below="@+id/imageLast"
android:layout_marginLeft="40dp"
android:layout_marginTop="20dp"
android:src="@drawable/icon_qrcode" />
```

Manually designed

```
<?xml version="1.0" encoding="utf−8"?>
<RelativeLayout xmlns:android="..."
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

<ImageButton
android:id="@+id/buttonOpen"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignParentLeft="true"
android:layout_alignParentTop="true"
android:layout_marginLeft="38dp"
android:layout_marginTop="55dp"
android:src="@drawable/icon_open" />

<ImageButton
android:id="@+id/buttonLast"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignLeft="@+id/buttonOpen"
android:layout_below="@+id/buttonOpen"
android:layout_marginTop="20dp"
android:src="@drawable/icon_recent" />

<ImageButton
android:id="@+id/buttonQRCode"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignLeft="@+id/buttonLast"
android:src="@drawable/icon_qrcode"
android:layout_below="@+id/buttonLast"
android:layout_marginTop="20dp"/>
```

The Welcome screen presents a kind of main menu in which every entry has an icon and, on the right of it, a text label. When the application starts, the user can 'touch' the icon of an entry, to perform the correspondent task. The manually developed application adopts the Android widget `ImageButton`, a 'touchable' image, that is not covered by the presented PIM. So, in the Platform Independent Model of the example application, the menu is realized placing side by side the icon with a labeled button, and this is translated in the Android PSM with the widgets `ImageView` and `Button`. The widget `Button` is used in place of the text label beside the icon. The listing on the right side shows that Android-specific advanced attributes, such as `alignParentLeft` and `alignParentTop` (and others not present in this ex-

ample), can be used to place the widget inside a screen, alternatively to those basic mapped by the transformation. Nevertheless, these advanced attributes cannot be abstracted and adopted in the PIM.

Auto-generated

```
<ImageView
android:id="@+id/imagePrefs"
android:layout_width="72dp"
android:layout_height="72dp"
android:layout_below="@+id/imageQRCode"
android:layout_marginLeft="40dp"
android:layout_marginTop="20dp"
android:src="@drawable/icon_prefs" />

<Button
android:id="@+id/buttonOpen"
android:layout_width="150dp"
android:layout_height="55dp"
android:layout_marginLeft="20dp"
android:layout_marginTop="45dp"
android:layout_toRightOf="@+id/imageOpen"
android:text="@string/wellcome_buttonOpen_text" />

<Button
android:id="@+id/buttonLast"
android:layout_width="150dp"
android:layout_height="55dp"
android:layout_below="@+id/buttonOpen"
android:layout_marginLeft="20dp"
android:layout_marginTop="40dp"
android:layout_toRightOf="@+id/imageLast"
android:text="@string/wellcome_buttonLast_text" />

<Button
android:id="@+id/buttonQRCode"
android:layout_width="150dp"
android:layout_height="55dp"
android:layout_below="@+id/buttonLast"
android:layout_marginLeft="20dp"
android:layout_marginTop="40dp"
android:layout_toRightOf="@+id/imageQRCode"
android:text="@string/wellcome_buttonQRCode_text" />

<Button
android:id="@+id/buttonPrefs"
android:layout_width="150dp"
android:layout_height="55dp"
android:layout_below="@+id/buttonQRCode"
android:layout_marginLeft="20dp"
android:layout_marginTop="40dp"
android:layout_toRightOf="@+id/imagePrefs"
android:text="@string/wellcome_buttonPrefs_text" />

</RelativeLayout>
```

Manually designed

```
<ImageButton
android:id="@+id/buttonPrefs"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignLeft="@+id/buttonQRCode"
android:layout_below="@+id/buttonQRCode"
android:layout_marginTop="20dp"
android:src="@drawable/icon_prefs" />

<TextView
android:id="@+id/label1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignTop="@+id/buttonOpen"
android:layout_centerHorizontal="true"
android:layout_marginTop="24dp"
android:text="@string/label1" />

<TextView
android:id="@+id/label2"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignLeft="@+id/label1"
android:layout_alignTop="@+id/buttonLast"
android:layout_marginTop="24dp"
android:text="@string/label2" />

<TextView
android:id="@+id/label3"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignLeft="@+id/label2"
android:layout_alignTop="@+id/buttonQRCode"
android:layout_marginTop="21dp"
android:text="@string/label3" />

<TextView
android:id="@+id/label4"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignLeft="@+id/label3"
android:layout_alignTop="@+id/buttonPrefs"
android:layout_marginTop="21dp"
android:text="@string/label4" />

</RelativeLayout>
```

Rule 14 (Menu Mapping): Transformation rule that generates the XML file that describes the structure of the menu that are available in a screen. The following parallel listings show the description of the menu available in the screen `NavigateMap`.

Auto-generated

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk
    /res/android" >
<item android:id="@+id/navigatemap_main_menu"
    android:title="@string/navigatemap_navmenu_text">
<menu>

<item android:id="@+id/navigatemap_taglistMenu"
    android:title="@string/navmenu_taglistMenu_text"/>
<item android:id="@+id/navigatemap_qrcodeMenu"
    android:title="@string/navmenu_qrcodeMenu_text"/>
</menu>
</item>
</menu>
```

Manually designed

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk
    /res/android" >
<item android:id="@+id/mainMenu"
    android:title="@string/menuLabelMain">
<menu>

<item android:id="@+id/menuTagList"
    android:title="@string/menuLabelTagList"/>
<item android:id="@+id/menuQRCode"
    android:title="@string/menuLabelQRCode"/>
</menu>
</item>
</menu>
```

The menu has two entries: `Acquire QR Code` and `Points of Interest list`. The former allows the visitor to acquire the information encoded in the QR Code, the latter shows to the user the list of the POI highlighted on the map.
Also in this case, the main difference is the standardized format of the attributes values (`android:title` and `android:id`). In the manually version the choice is up to the developer, and it may not follow a standard schema.

Code Generation With regard to the generation of the application code, in Appendix C the following listings are reported:

- Listing C.1 - Android resource file `R.java`
  This is the file that collects all the resources of the application (widgets, layouts, menu, strings and images) and assigns to them a numeric constant. This file is substantially the same as the manually generated, except for the values of the assigned constants.

- Listing C.2 - The auto-generated java file of the activity `Wellcome`
  This file can be compared with the Listing C.3, that reports the manually developed version. The main difference is the number of methods and variables added to the class. In fact, to handles the 'touch' event of the widgets, the code-generation rules create, for each widget composing the Activity and that may trigger the event, a private method with the suffix `_Click`. The invocation of the correct method is handled by the `switch` statement in the private method `onClick()`. Furthermore, for each widget composing the Activity, the rules add a private variable to the class. This variable serves as a pointer to the widget and it is initialized even if it is not necessary and it will never used. The manually developed file is more compact, because uses a smaller number of variables and handles the 'touch' events directly in the private method `onClick()`.

- Listing C.4 - The auto-generated java file of the activity `TagList`
  This file can be compared with the Listing C.5, that reports the manually developed version. In this file we analyze the generated code in the presence of the ListBox widget. The code-generation rules add, for each ListBox widget composing the Activity, the private method with suffix `_ItemClick` to the class. The `switch` statement, in the private method `onItemClick`, allows to invoke the correct method handler for the widget that has trigged the `onItemClick` event. This is necessary because, implementing the interface `OnItemClickListener`, only one main `onItemClick` event handler can be used. In the manually developed file, whenever there is only one Listbox widget, the additional method `_ItemClick` may be omitted.

The examined listings cover the most interesting cases. In all the other cases, the auto-generated code conforms to what should be developed manually.

Finally, the Listing C.6 shows the auto-generated code of the activity `NavigateMap`. This activity implements a Menu and the redirection of the *Back Button* to the `Wellcome` screen.

6.3.2 Windows Phone 7

Structure Mapping transformation rule generates the class diagram depicted in Figure B.6. The Class Diagram shows the seven Screen classes that in Windows Phone must specialize the class `PhoneApplicationPage`, and the interfaces implemented by some of them.

WMAppManifest.xml is the manifest file of the application, generated by the rule **Manifest File**. The code is the same of the manually developed.

```
<?xml version="1.0" encoding="utf-8"?>
<Deployment
  xmlns="http://schemas.microsoft.com/windowsphone/2009/deployment"
  AppPlatformVersion="7.0">
  <App xmlns="" Title="MapNavigator" RuntimeType="Silverlight"
       Version="1" Genre="apps.normal"
       Author="" Description="" Publisher="">
    <IconPath IsRelative="true" IsResource="false">ApplicationIcon.png</IconPath>
  <Capabilities>
    <Capability Name="ID_CAP_NETWORKING"/>
    <Capability Name="ID_HW_FRONTCAMERA"/>
    <Capability Name="ID_CAP_CAMERA"/>
    <Capability Name="ID_CAP_ISV_CAMERA"/>
    <Capability Name="ID_CAP_MEDIALIB"/>
  </Capabilities>
  <Tasks>
    <DefaultTask Name =" _default" NavigationPage="Wellcome.xaml"/>
  </Tasks>
  <Tokens>
    <PrimaryToken TokenID="MapNavigatorToken" TaskName="_default">
      <TemplateType5>
        <BackgroundImageURI IsRelative="true"
                    IsResource="false">
          Background.png
        </BackgroundImageURI>
        <Count>0</Count>
        <Title>MapNavigator</Title>
      </TemplateType5>
    </PrimaryToken>
  </Tokens>
</App></Deployment>
```

In the first part of the manifest there are the `<Capability>` tags that map the attribute `resources` of the PIM class `Application`, by the transformation rule **ResourcePermissions Mapping** .

GUI Layout The following listings show, in order, the most significant part of the XAML file obtained by the transformation rules and the same XAML file manually designed. These files describe the GUI Layout of the screen `Preferences`, and they are reported in the Listings C.7 and C.8

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="0,0,0,0">
  <Image Height="108" Width="108" Margin="30,30,0,0"
    Name="imageIcon" Source="icon_prefs.png"/>
  <TextBlock Height="38" Width="150" Margin="168,75,0,0"
    Name="labelTitle" Text="Preferences" />
  <TextBlock Height="30" Width="225" Margin="30,168,0,0"
    Name="labelLocation" Text="Default_maps_location"/>
  <TextBox Height="80" Width="420" Margin="30,206,0,0"
    Name="txtLocation" Text=""/>
  <CheckBox Height="53" Width="300" Margin="30,301,0,0"
    Content="Save_the_latest_map" Name="checkLast" Click="checkLast_Click"/>
  <CheckBox Height="53" Width="300" Margin="30,369,0,0"
    Content="Search_for_updates" Name="checkUpdate" Click="checkUpdate_Click"/>
  <Button Height="83" Width="225" Margin="144,600,0,0"
    Content="Save" Name="buttonSave" Click="buttonSave_Click" />
</Grid>
```

```
<!--ContentPanel - place additional content here-->
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="0,0,0,0">
  <TextBlock Height="37" HorizontalAlignment="Left" Margin="28,132,0,0"
        Name="label2" Text="Default_maps_location"
        VerticalAlignment="Top" Width="206" />
  <TextBox Height="70" HorizontalAlignment="Left" Margin="14,154,0,0"
      Name="txtLocation" Text="" VerticalAlignment="Top" Width="456" />
  <CheckBox Content="Save_the_last_opened_map" Height="72"
      HorizontalAlignment="Left" Margin="14,230,0,0"
      Name="checkLastMap" VerticalAlignment="Top" Width="353"
      Click="checkLastMap_Click" />
  <CheckBox Content="Search_for_update" Height="72" HorizontalAlignment="Left"
      Margin="12,291,0,0" Name="checkUpdate" VerticalAlignment="Top"
      Width="353" Click="checkUpdate_Click" />
  <Image Height="79" HorizontalAlignment="Left" Margin="25,22,0,0"
      Name="imageIcon" Stretch="Fill" VerticalAlignment="Top" Width="88"
      Source="/MapNavigator;component/icon_prefs.png" />
  <TextBlock Height="35" Margin="153,45,204,0" Name="label1"
      Text="Preferences" VerticalAlignment="Top" />
  <Button Content="Save" Height="68" HorizontalAlignment="Left"
      Margin="114,369,0,0" Name="buttonSave" VerticalAlignment="Top"
      Width="253" Click="buttonSave_Click" />
</Grid>
```

The values of the attributes `Height` `Width` and `Margin` are obtained from the PIM values by multiplying the scale factor 1.5, as explained in section 5.5.

The second listing shows that XAML-specific advanced attributes, such as `HorizontalAlignment` and `VerticalAlignment` (and others not present in this example), can be used to place the widget inside a screen, alternatively to those basic mapped by the transformation. Nevertheless, these advanced attributes cannot be abstracted and adopted in the PIM. The transformation rules can only use the platform-specific attributes that have a counterpart in the PIM.

Menu Mapping transformation rule inserts, in the XAML file of the page providing the menu, the structure description of each submenu. The following listing shows the description of the menu available in the screen `NavigateMap`.

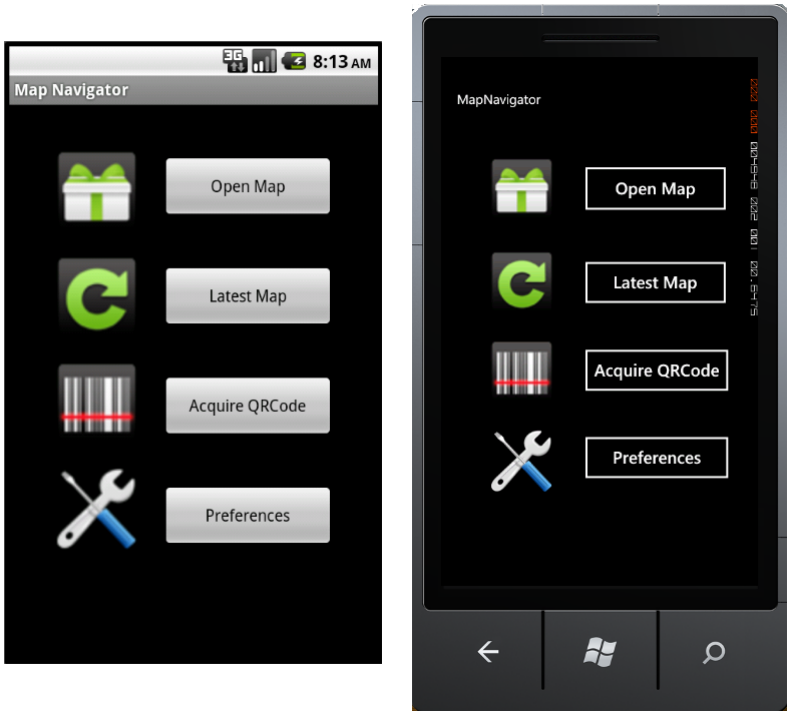
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
<item android:id="@+id/navigatemap_main_menu"
    android:title="@string/navigatemap_navmenu_text">
<menu>

<item android:id="@+id/navigatemap_taglistMenu"
    android:title="@string/navmenu_taglistMenu_text"/>
<item android:id="@+id/navigatemap_qrcodeMenu"
    android:title="@string/navmenu_qrcodeMenu_text"/>
</menu>
</item>
</menu>
```

Also in the menu description in the Windows Phone XAML files, the main difference is the standardized format of the value of the attribute `Click`. In the manually version the choice is up to the developer, and it may not follow a standard schema.

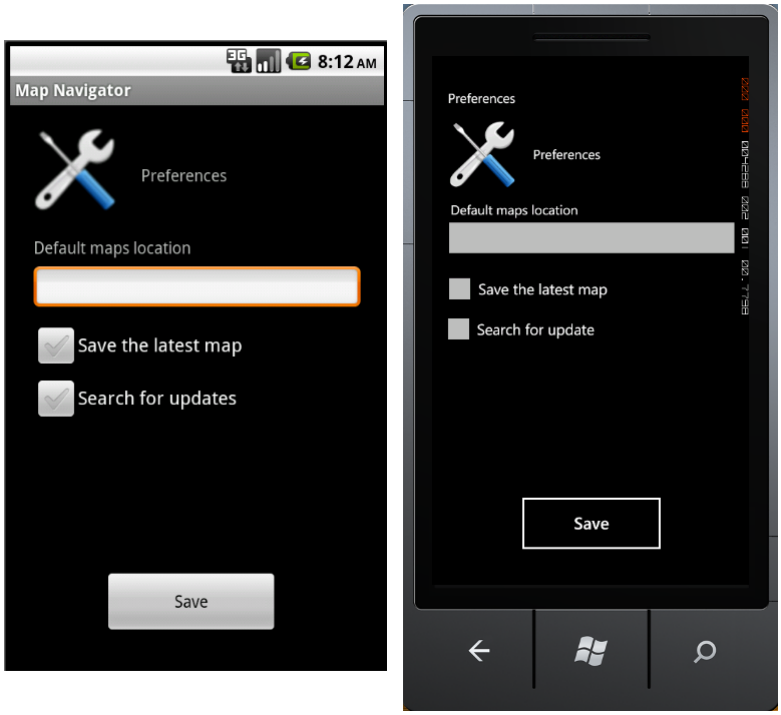
Code Generation The Listings C.9 and C.10 reports the code of the two partial classes for the page `Preferences`. In Windows Phone the difference between auto-generated code and the manually developed code is minimal, due to the different framework provided by the eXtensible Application Markup Language. In fact, in the XAML file that describe the Pages GUI layout, the widgets tags can specify which class method handles the events that they trigger. So, in the class file the developer must only insert the relative event handlers with the appropriate signatures. On the contrary of Android, there is no need to insert code to initialize the layout and the menu. The XAML file describe all these elements and the underlying Silverlight framework initializes them properly.

6.3.3 Graphical Comparison

Finally, the obtained GUI layout for both Android and Windows Phone can be compared. Figure 6.2 shows the comparison of two screens of the application. On the left side it reports the Android version, and on the right side the Windows Phone version. The two layouts are very similar, and they show how the widgets, abstracted and usable in the PIM, appear on each target platform.



(a) Wellcome Screen



(b) Preferences Screen

Figure 6.2: Comparison of the GUI Layout

Chapter 7

Conclusions

The main goal of the thesis has been the study of a model-driven approach to the design of multi-platform smartphone applications. In order to achieve this goal, it has been necessary to analyze the major smartphone platforms on the market and their software development environments. The analysis has led to highlight the common aspects between different platforms and different modes of development. In every development kit, the description of the Graphical User Interface is separated from the application code. Furthermore, every platform requires a "manifest" file that defines some fundamental characteristics of the application, to be provided to the user and to the system during the installation phase. A smartphone application involves several number of screens, composed by a set of widgets, with which users can interact in order to perform several tasks.

The set of widgets provided by the smartphone platforms may be wide, but all platforms shares a common and standardized subset of them. Then, the Platform Independent Model has been defined. It allows to design smartphone applications independently by a specific platform. The proposed UML2 Profile for the PIM provides all the fundamental elements (widget classes, enumerations, and stereotypes) to design applications using three diagrams: Class Diagram, Object Diagram and Statechart Diagram.

According to the Model Driven Architecture, two Platform-Specific Models have been described for both Android and Windows Phone platforms, together with a set of detailed model transformations. Also these models use Class Diagram and Statechart Diagram, together with specific metamodels to describe the GUI Layout of the screens composing application.

Finally, to complete the model-driven process, a set of code-generation rules has been created to produce the application code for the two examined platforms. The code that the rules are able to generate covers the structural aspects of the application, avoiding the developer to write several repetitive code. This code represents the structure of the main classes (one for each screen composing the application) and some initialization code for the GUI components (widgets, menu and event handlers). Furthermore, the rules generate code that implement behavioral aspects, e.g. screen transitions, according to those defined in the Statechart Diagram. In the traditional approach, this auto-generated code should be manually written several times.

The result covers the entire process of model-driven design, and identifies the opportunity to develop cross-platform smartphone applications, starting from a platform-independent model, without the use of middleware that interpret a high-level description of the GUI, in every platform.

In the end, Chapter 6 introduces MapNavigator, a sample application useful for the experimental validation. The chapter presents the Platform Independent Model of MapNavigator, and discusses the application of the transformation and code-generation rules. Some evaluation metrics have been introduced, in order to compare the obtained result with the same application developed in the traditional way.

7.1 Future Work

The experimental validation shows a great result in comparison between the auto-generated code and the manual implementation for each target platform. The obtained GUI is very similar for the two examined platforms, but they cannot benefit from their specific set of advanced widgets. The proposed Platform Independent Model shows an example of which can be abstracted from the major smartphone platforms, but it could be improved and expanded with more common features (in particular widgets).

The transformation rules have not been implemented in a specific transformation language, due to the complexity to act directly on the entire UML Meta-model, and to maintain the methodology independent from a specific implementation. A possible future work is the definition, for the Platform Independent Model, of a custom metamodel representing the language to "talk about" smartphone applications. This metamodel should be constructed

as an instance of M2-level shown in figure 3.5, and it should provide high-level elements, such as screens, widgets and a state machine with transitions between the screens. Such a model would be easier than the entire UML metamodel (that have a huge number of basic elements) and it would be more easily manipulated by the transformation language. However, with this simplified model you lose the ability to design all the other aspects of the application, not specific for a smartphone. These aspects, concerning classes, interfaces, operations, and attributes, realize the other specific functionality of the application to be modeled.

Another aspect to be addressed in detail is the translation of the OCL boolean expressions, defining the transition guards in the PIM Statechart Diagram, into Java or C-Sharp code as described in the code-generation rules **Guards on Transitions**. Existing tools capable of translating OCL expressions cover the entire set of OCL constructs [9, 10]. Less complex techniques should be developed, to easily translate the guards, represented only by boolean expressions.

Appendix A

Tables

Table A.1: List of the Classes in UML Profile for Platform Independent Model

Name	**Attributes and Operations**
Screen	title: String orientation: OrientationEnum fullScreen: Boolean widgets: Widget[1..*] layout: ScreenLayoutEnum
Menu	title: String
Submenu	title: String style: ItemStyleEnum checked: Boolean icon: String
Application	title: String version: Integer resources: ResourcesEnum [0..*]
Widget	width: integer height: integer above: Widget[0..1] below: Widget[0..1] toLeftOf: Widget[0..1] toRightOf: Widget[0..1] marginLeft: integer marginRight: integer marginTop: integer marginBottom: integer
Label	text: String
Textbox	text: String
Button	text: String
Checkbox	text: String
Imagebox	imageSource: String
Progressbar	maxValue: integer getCurrentValue(): integer
Listbox	populate(items:Listbox[1..*])
ListboxItem	text: String
AbstractDevice	displayWidth: integer displayHeight: integer

Appendix B

Figures

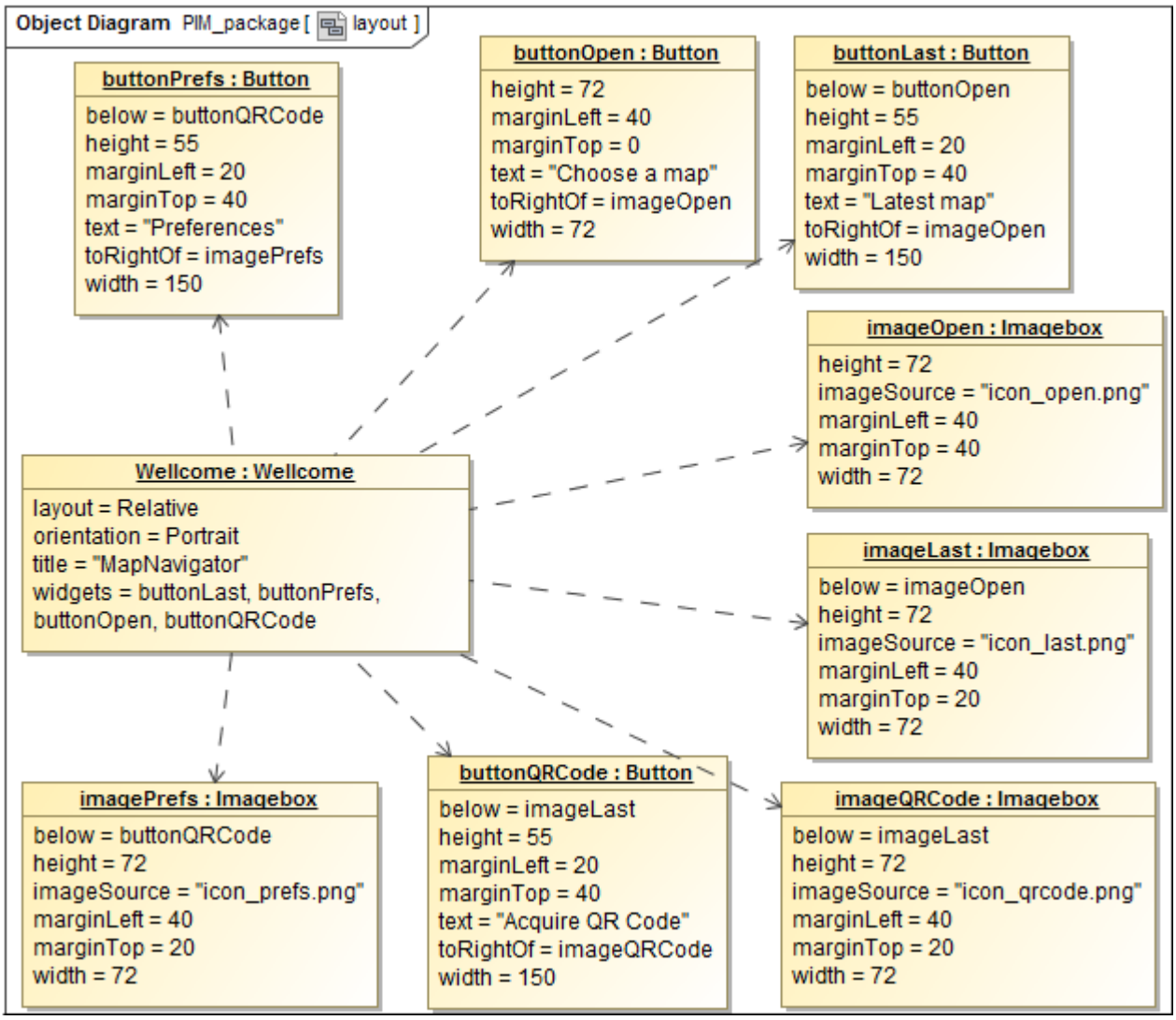


Figure B.1: PIM Object Diagram - part 1

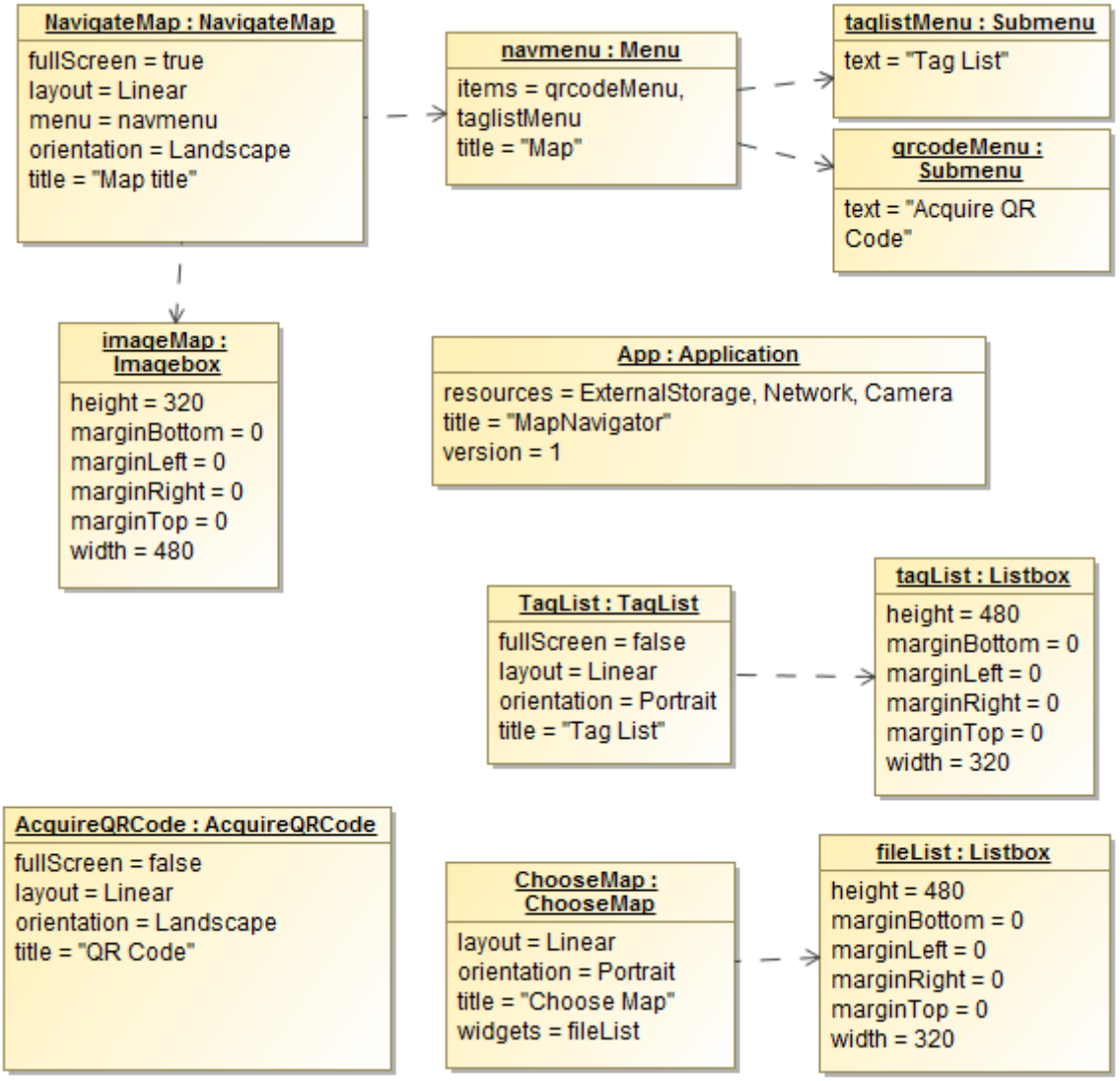


Figure B.2: PIM Object Diagram - part 2

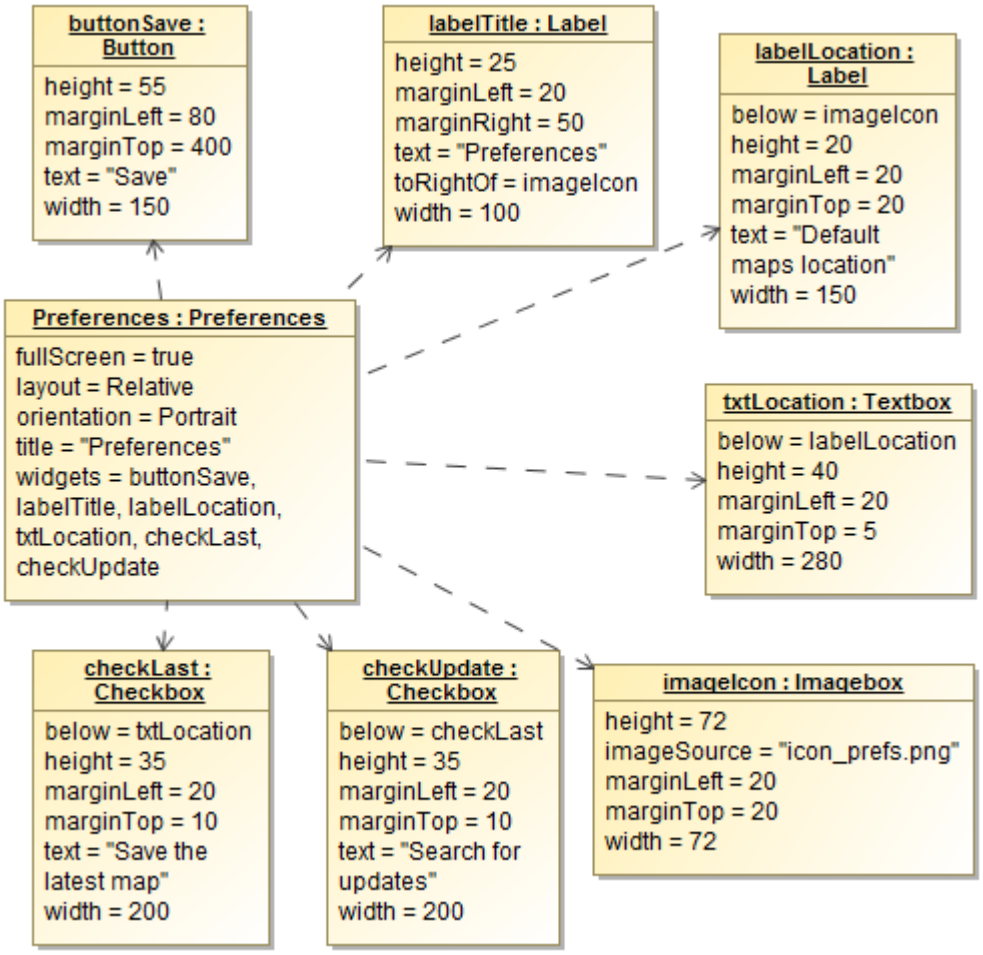


Figure B.3: PIM Object Diagram - part 3

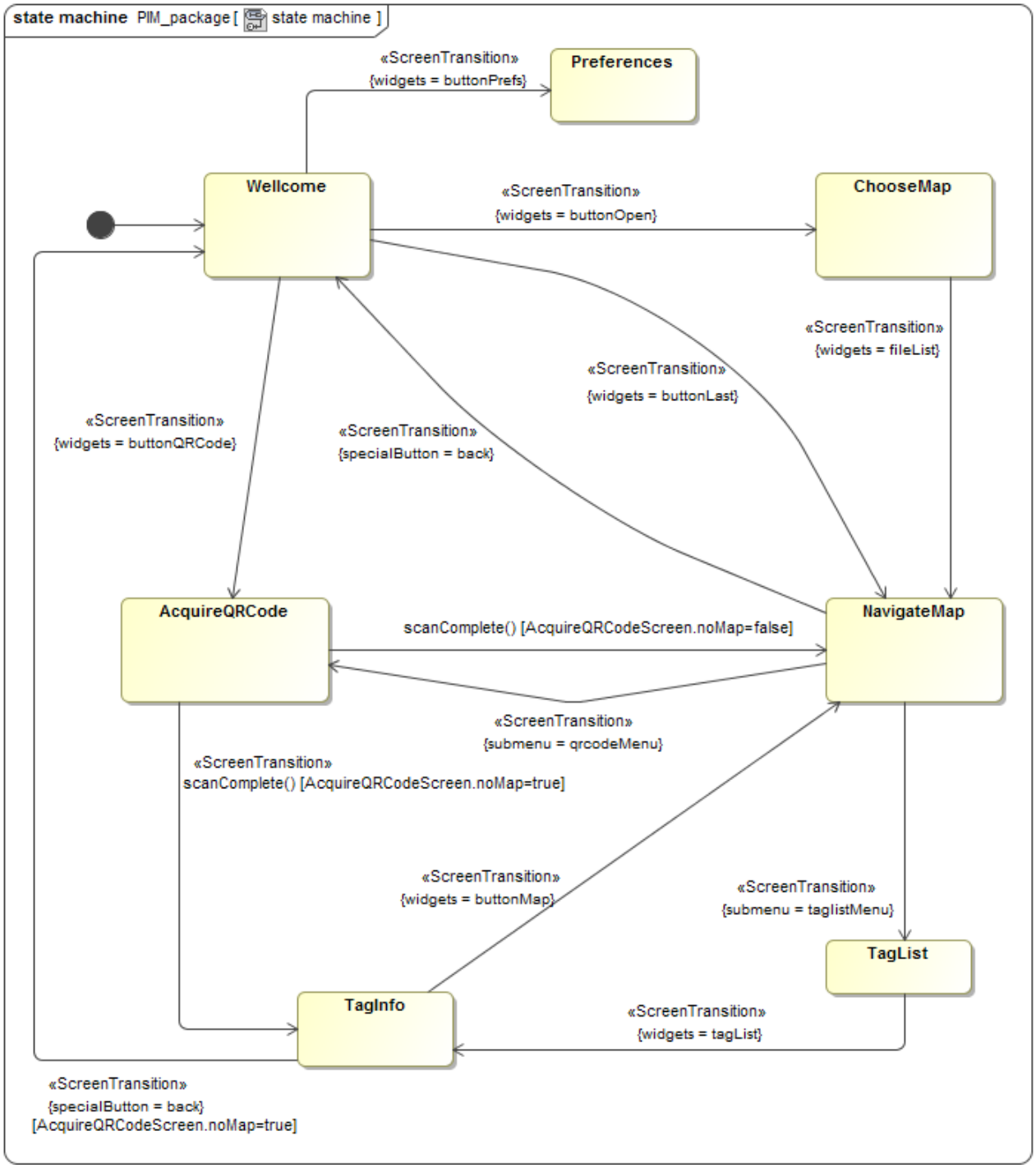


Figure B.4: PIM - Statechart Diagram

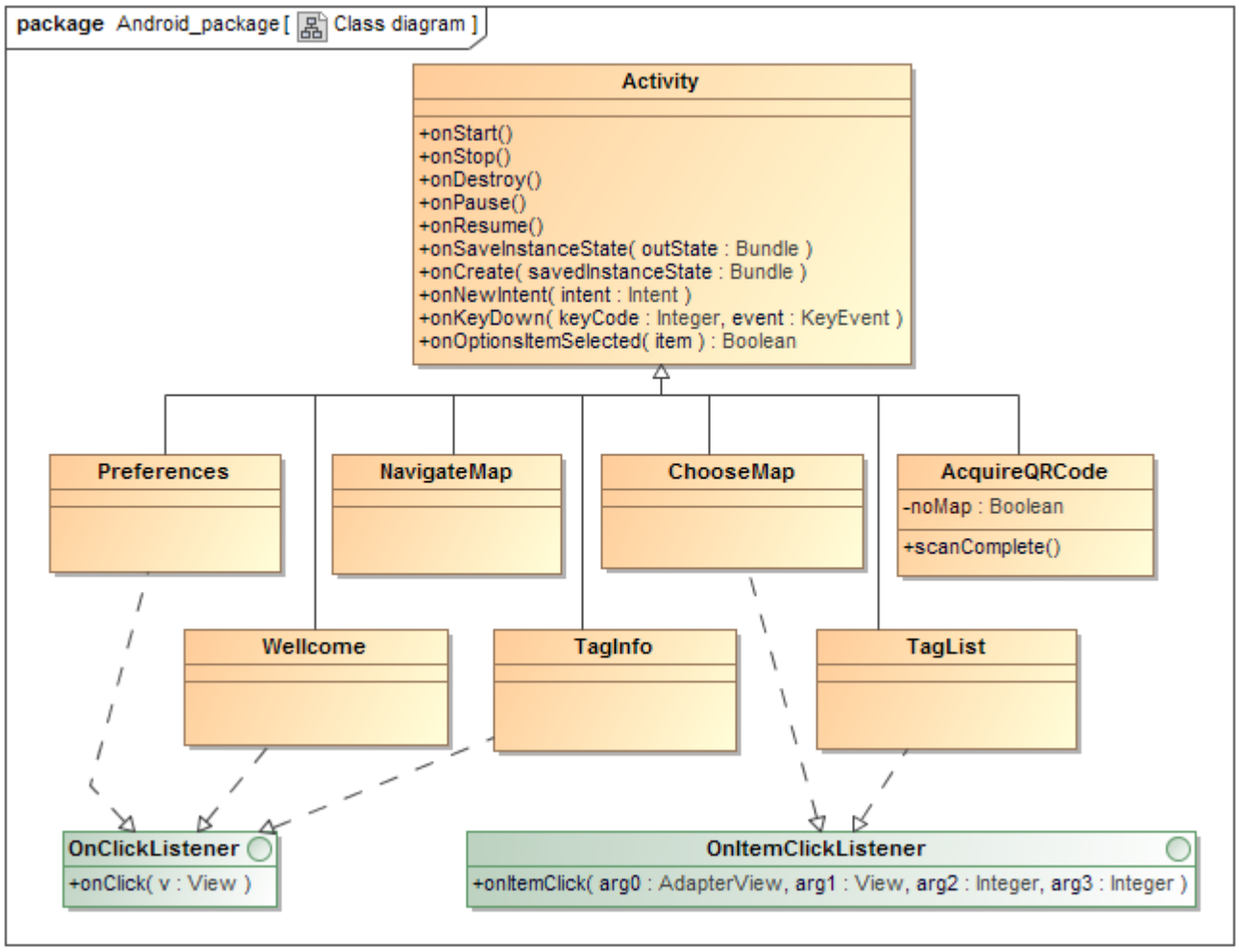


Figure B.5: Android PSM - Class Diagram

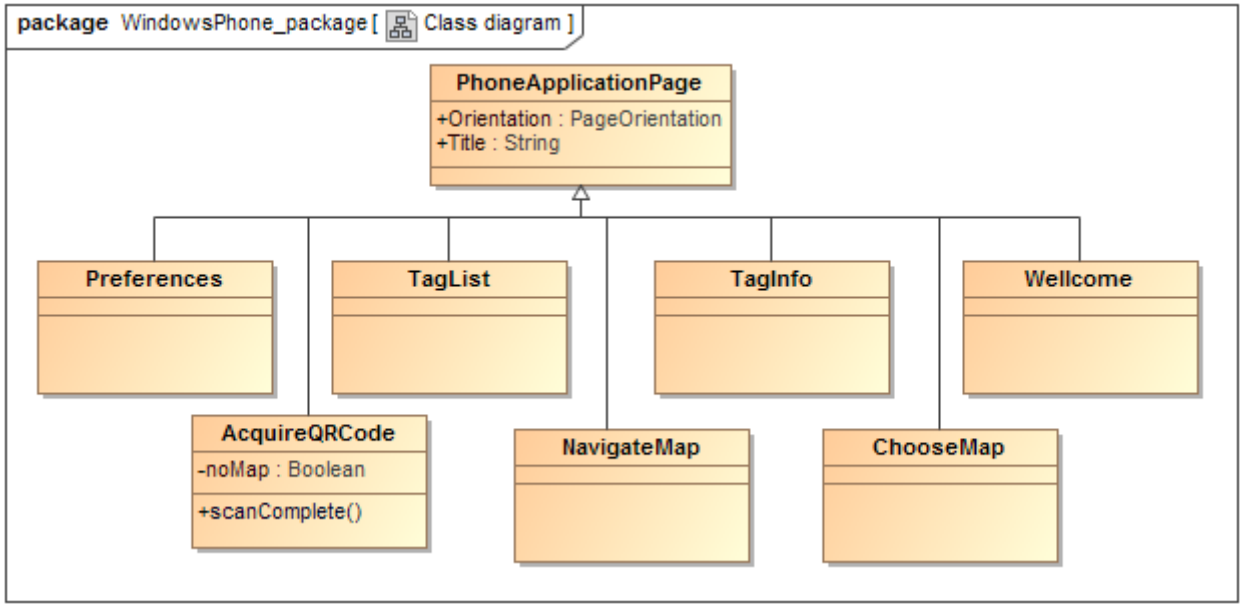


Figure B.6: Windows Phone PSM - Class Diagram

Appendix C

Code Listings

Listing C.1: Auto-generated Android resources file `R.java`

```
1  package univr.MapNavigator;
2  public final class R {
3      public static final class string {
4          public static final int app_name=0x7f0;
5          public static final int navigatemap_navmenu_text=0x7f1;
6          public static final int navmenu_qrcodeMenu_text=0x7f2;
7          public static final int navmenu_taglistMenu_text=0x7f3;
8          public static final int preferences_buttonSave_text=0x7f4;
9          [...]
10     }
11   public static final class menu {
12         public static final int navmenu=0x7f22;
13     }
14     public static final class layout {
15         public static final int preferences=0x7f23;
16         public static final int wellcome=0x7f24;
17         [...]
18     }
19     public static final class drawable {
20         public static final int ic_launcher=0x7f30;
21         public static final int icon_help=0x7f31;
22         public static final int icon_open=0x7f32;
23         [...]
24     }
25     public static final class id {
26         public static final int buttonLast=0x7f37;
27         public static final int buttonOpen=0x7f38;
28         public static final int buttonPrefs=0x7f39;
29         [...]
30     }
31 }
```

Listing C.2: Auto-generated java file of the `Wellcome` Activity

```
1  package univr.MapNavigator.activity;
2  import univr.MapNavigator.R;
3  import android.app.Activity;
4  import android.content.Context;
5  import android.content.Intent;
6  import android.view.View;
7  import android.view.View.OnClickListener;
8  import android.widget.AdapterView.OnItemClickListener;
9  import android.os.Bundle;
10 import android.widget.*;
11
12 public class Wellcome extends Activity implements OnClickListener {
13   private Button buttonOpen;
14   private Button buttonLast;
15   private Button buttonQRCode;
16   private Button buttonPrefs;
17   private ImageView imageOpen;
18   private ImageView imageLast;
19   private ImageView imageQRCode;
20   private ImageView imagePrefs;
21
22     public void onCreate(Bundle savedInstanceState) {
23       super.onCreate(savedInstanceState);
24         setContentView(R.layout.wellcome);
25     setTitle("Map_Navigator");
26
27     buttonOpen = (Button) findViewById(R.id.buttonOpen);
28     buttonLast = (Button) findViewById(R.id.buttonLast);
29     buttonQRCode = (Button) findViewById(R.id.buttonQRCode);
30     buttonPrefs = (Button) findViewById(R.id.buttonPrefs);
31     imageOpen = (ImageView) findViewById(R.id.imageOpen);
32     imageLast = (ImageView) findViewById(R.id.imageLast);
33     imageQRCode = (ImageView) findViewById(R.id.imageQRCode);
34     imagePrefs = (ImageView) findViewById(R.id.imagePrefs);
35
36     buttonOpen.setOnClickListener(this);
37     buttonLast.setOnClickListener(this);
38     buttonQRCode.setOnClickListener(this);
39     buttonPrefs.setOnClickListener(this);
40     }
41   public void onClick(View v) {
42     switch (v.getId()) {
43       case R.id.buttonOpen:
44         buttonOpen_Click();
45         break;
46       case R.id.buttonLast:
47         buttonLast_Click();
48         break;
```

```
49          case R.id.buttonQRCode:
50            buttonQRCode_Click();
51            break;
52          case R.id.buttonPrefs:
53            buttonPrefs_Click();
54            break;
55        }
56      }
57      private void buttonOpen_Click() {
58        Intent intentOpen = new Intent(this, ChooseMap.class);
59        startActivity(intentOpen);
60      }
61      private void buttonQRCode_Click() {
62        Intent intentQRCode = new Intent(this, AcquireQRCode.class);
63        startActivity(intentQRCode);
64      }
65      private void buttonLast_Click() {
66        Intent intentQRCode = new Intent(this, NavigateMap.class);
67        startActivity(intentQRCode);
68      }
69      private void buttonPrefs_Click() {
70        Intent intentQRCode = new Intent(this, Preferences.class);
71        startActivity(intentQRCode);
72      }
73  }
```

Listing C.3: Manually developed java file of the `Wellcome` Activity

```
1  package univr.mapnavigator.activity;
2
3  import univr.mapnavigator.R;
4  import android.app.Activity;
5  import android.content.Intent;
6  import android.os.Bundle;
7  import android.view.View;
8  import android.view.View.OnClickListener;
9  import android.widget.ImageButton;
10
11 public class MainActivity extends Activity implements OnClickListener
12 {
13   private ImageButton buttonOpen;
14   private ImageButton buttonLast;
15   private ImageButton buttonQRCode;
16   private ImageButton buttonPrefs;
17
18     @Override
19     public void onCreate(Bundle savedInstanceState)
20     {
21       super.onCreate(savedInstanceState);
22         setContentView(R.layout.main);
23     setTitle("Map Navigator");
24
25     buttonOpen = (ImageButton) findViewById(R.id.buttonOpen);
26     buttonLast = (ImageButton) findViewById(R.id.buttonLast);
27     buttonQRCode = (ImageButton) findViewById(R.id.buttonQRCode);
28     buttonPrefs = (ImageButton) findViewById(R.id.buttonPrefs);
29
30     buttonOpen.setOnClickListener(this);
31     buttonLast.setOnClickListener(this);
32     buttonQRCode.setOnClickListener(this);
33     buttonPrefs.setOnClickListener(this);
34     }
35
36   public void onClick(View v)
37   {
38     switch (v.getId())
39     {
40       case R.id.buttonOpen:
41         Intent intentOpen = new Intent(this, ChooseMapActivity.class);
42         startActivity(intentOpen);
43         break;
44       case R.id.buttonQRCode:
45         Intent intentQRCode = new Intent(this, NavigateMapActivity.class);
46         startActivity(intentQRCode);
47         break;
48       case R.id.buttonPrefs:
```

```
49            Intent intentPrefs = new Intent(this, PreferencesActivity.class);
50            startActivity(intentPrefs);
51            break;
52        }
53    }
54 }
```

Listing C.4: Auto-generated java file of the `TagList` Activity

```
1  package univr.MapNavigator.activity;
2
3  import univr.MapNavigator.R;
4  import android.app.Activity;
5  import android.content.Context;
6  import android.content.Intent;
7  import android.view.View;
8  import android.view.View.OnClickListener;
9  import android.widget.AdapterView.OnItemClickListener;
10 import android.os.Bundle;
11 import android.widget.*;
12
13 public class TagList extends Activity implements OnItemClickListener {
14   private ListView tagList;
15
16     public void onCreate(Bundle savedInstanceState) {
17       super.onCreate(savedInstanceState);
18         setContentView(R.layout.taglist);
19     setTitle("Tag_List");
20
21     tagList = (ListView) findViewById(R.id.tagList);
22     tagList.setOnItemClickListener(this);
23     }
24
25   public void onItemClick(AdapterView<?> arg0, View arg1, int arg2, long arg3) {
26     Intent intent;
27     switch (arg0.getId()) {
28       case R.id.tagList:
29         tagList_ItemClick(arg2);
30         break;
31     }
32   }
33
34   private void tagList_ItemClick(int item) {
35     Intent intentQRCode = new Intent(this, TagInfo.class);
36     startActivity(intentQRCode);
37   }
38 }
```

Listing C.5: Manually developed java file of the `TagList` Activity

```
1  package univr.mapnavigator.activity;
2
3  import univr.mapnavigator.R;
4  import android.app.Activity;
5  import android.content.Intent;
6  import android.os.Bundle;
7  import android.view.View;
8  import android.widget.AdapterView;
9  import android.widget.AdapterView.OnItemClickListener;
10 import android.widget.ListView;
11
12 public class TagListActivity extends Activity implements OnItemClickListener
13 {
14   private ListView tagList;
15
16     @Override
17     public void onCreate(Bundle savedInstanceState)
18     {
19       super.onCreate(savedInstanceState);
20         setContentView(R.layout.taginfo);
21     setTitle("Map Navigator");
22
23     tagList = (ListView) findViewById(R.id.fileList);
24     tagList.setOnItemClickListener(this);
25     }
26
27   public void onItemClick(AdapterView<?> arg0, View viewId, int itemId, long arg3
         )
28   {
29     if (arg0.getId() == R.id.tagList)
30     {
31       // ...
32       Intent intentQRCode = new Intent(this, TagInfo.class);
33       startActivity(intentQRCode);
34     }
35   }
36 }
```

Listing C.6: Auto-generated java file of the `NavigateMap` Activity

```
1  package univr.MapNavigator.activity;
2
3  import univr.MapNavigator.R;
4  import android.app.Activity;
5  import android.content.Intent;
6  import android.os.Bundle;
7  import android.view.KeyEvent;
8  import android.view.Menu;
9  import android.view.MenuInflater;
10 import android.view.MenuItem;
11
12 public class NavigateMap extends Activity
13 {
14     public void onCreate(Bundle savedInstanceState)
15     {
16       super.onCreate(savedInstanceState);
17         setContentView(R.layout.navigatemap);
18     setTitle("Map_title");
19     }
20
21   public boolean onCreateOptionsMenu(Menu menu)
22   {
23     MenuInflater inflater = getMenuInflater();
24     inflater.inflate(R.menu.navmenu , menu);
25     return true;
26   }
27
28   public boolean onKeyDown(int keyCode, KeyEvent event)
29   {
30     switch(keyCode)
31     {
32       case KeyEvent.KEYCODE_BACK:
33         Intent intentn = new Intent(this, Wellcome.class);
34         startActivity(intent);
35         break;
36     }
37     return super.onKeyDown(keyCode, event);
38     }
39
40   public boolean onOptionsItemSelected(MenuItem item)
41   {
42     switch (item.getItemId())
43     {
44       case R.id.navigatemap_taglistMenu:
45         Intent intentOpen = new Intent(this, TagList.class);
46         startActivity(intentOpen);
47         break;
48
```

```
49          case R.id.navigatemap_qrcodeMenu:
50            Intent intentOpen = new Intent(this, AcquireQRCode.class);
51            startActivity(intentOpen);
52            break;
53        }
54        return true;
55      }
56  }
```

Listing C.7: Auto-generated XAML file of the page `Preferences`

```
<phone:PhoneApplicationPage
    x:Class="MapNavigator.Preferences"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    mc:Ignorable="d" d:DesignHeight="768" d:DesignWidth="480"
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,0">
        <TextBlock x:Name="PageTitle" Text="Preferences"
                   Style="{StaticResource PhoneTextNormalStyle}"/>
    </StackPanel>
  <Grid x:Name="ContentPanel" Grid.Row="1" Margin="0,0,0,0">
    <Image Height="108" Width="108" Margin="30,30,0,0"
      Name="imageIcon" Source="icon_prefs.png"/>
    <TextBlock Height="38" Width="150" Margin="168,75,0,0"
      Name="labelTitle" Text="Preferences" />
    <TextBlock Height="30" Width="225" Margin="30,168,0,0"
      Name="labelLocation" Text="Default_maps_location"/>
    <TextBox Height="80" Width="420" Margin="30,206,0,0"
      Name="txtLocation" Text=""/>
    <CheckBox Height="53" Width="300" Margin="30,301,0,0"
      Content="Save_the_latest_map" Name="checkLast" Click="checkLast_Click"
          />
    <CheckBox Height="53" Width="300" Margin="30,369,0,0"
      Content="Search_for_updates" Name="checkUpdate" Click="
          checkUpdate_Click"/>
    <Button Height="83" Width="225" Margin="144,600,0,0"
      Content="Save" Name="buttonSave" Click="buttonSave_Click" />
  </Grid>
    </Grid>
</phone:PhoneApplicationPage>
```

Listing C.8: Manually designed XAML file of the page `Preferences`

```
<phone:PhoneApplicationPage
    x:Class="MapNavigator.Preferences"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    mc:Ignorable="d" d:DesignHeight="768" d:DesignWidth="480"
    shell:SystemTray.IsVisible="True">

    <!--LayoutRoot is the root grid where all page content is placed-->
    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,0">
            <TextBlock x:Name="PageTitle" Text="Preferences"
                       Style="{StaticResource PhoneTextNormalStyle}"/>
        </StackPanel>
        <!--ContentPanel - place additional content here-->
        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="0,0,0,0">
            <TextBlock Height="37" HorizontalAlignment="Left" Margin="
                28,132,0,0"
                       Name="label2" Text="Default maps location"
                       VerticalAlignment="Top" Width="206" />
            <TextBox Height="70" HorizontalAlignment="Left" Margin="
                14,154,0,0"
                     Name="txtLocation" Text="" VerticalAlignment="Top"
                         Width="456" />
            <CheckBox Content="Save the last opened map" Height="72"
                      HorizontalAlignment="Left" Margin="14,230,0,0"
                      Name="checkLastMap" VerticalAlignment="Top" Width="
                          353"
                      Click="checkLastMap_Click" />
            <CheckBox Content="Search for update" Height="72"
                HorizontalAlignment="Left"
                      Margin="12,291,0,0" Name="checkUpdate"
                           VerticalAlignment="Top"
                      Width="353" Click="checkUpdate_Click" />
            <Image Height="79" HorizontalAlignment="Left" Margin="
                25,22,0,0"
```

```
41                      Name="imageIcon" Stretch="Fill" VerticalAlignment="Top"
                            Width="88"
42                      Source="/MapNavigator;component/icon_prefs.png" />
43                <TextBlock Height="35" Margin="153,45,204,0" Name="label1"
44                        Text="Preferences" VerticalAlignment="Top" />
45                <Button Content="Save" Height="68" HorizontalAlignment="Left"
46                        Margin="114,369,0,0" Name="buttonSave" VerticalAlignment
                            ="Top"
47                        Width="253" Click="buttonSave_Click" />
48            </Grid>
49        </Grid>
50    </phone:PhoneApplicationPage>
```

Listing C.9: Partial class of the page `Preferences`, obtained by Code-Generation

```
1  using System;
2  using System.Collections.Generic;
3  using System.Windows;
4  using System.Windows.Controls.Primitives;
5  using System.Windows.Controls;
6  using System.Windows.Documents;
7  using System.Windows.Input;
8  using System.Windows.Media;
9  using System.Windows.Media.Animation;
10 using System.Windows.Shapes;
11 using Microsoft.Phone.Controls;
12 using System.Windows.Threading;
13
14 namespace MapNavigator {
15     public partial class Preferences : Microsoft.Phone.Controls.PhoneApplicationPage
           {
16         internal System.Windows.Controls.Grid LayoutRoot;
17         internal System.Windows.Controls.StackPanel TitlePanel;
18         internal System.Windows.Controls.TextBlock PageTitle;
19         internal System.Windows.Controls.Grid ContentPanel;
20         internal System.Windows.Controls.Image imageIcon;
21         internal System.Windows.Controls.TextBlock labelTitle;
22         internal System.Windows.Controls.TextBlock labelLocation;
23         internal System.Windows.Controls.TextBox txtLocation;
24         internal System.Windows.Controls.CheckBox checkLast;
25         internal System.Windows.Controls.CheckBox checkUpdate;
26         internal System.Windows.Controls.Button buttonSave;
27
28         private bool _contentLoaded;
29
30         [System.Diagnostics.DebuggerNonUserCodeAttribute()]
31         public void InitializeComponent() {
32             if (_contentLoaded) {
33                 return;
34             }
35             _contentLoaded = true;
36             System.Windows.Application.LoadComponent(this, new System.Uri("/
                   MapNavigator;component/Preferences.xaml", System.UriKind.Relative)
                   );
37             this.LayoutRoot = ((System.Windows.Controls.Grid)(this.FindName("
                   LayoutRoot")));
38             this.TitlePanel = ((System.Windows.Controls.StackPanel)(this.FindName(
                   "TitlePanel")));
39             this.PageTitle = ((System.Windows.Controls.TextBlock)(this.FindName("
                   PageTitle")));
```

```
40              this.ContentPanel = ((System.Windows.Controls.Grid)(this.FindName("
                    ContentPanel")));
41              this.imageIcon = ((System.Windows.Controls.Image)(this.FindName("
                    imageIcon")));
42              this.labelTitle = ((System.Windows.Controls.TextBlock)(this.FindName("
                    labelTitle")));
43              this.labelLocation = ((System.Windows.Controls.TextBlock)(this.
                    FindName("labelLocation")));
44              this.txtLocation = ((System.Windows.Controls.TextBox)(this.FindName("
                    txtLocation")));
45              this.checkLast = ((System.Windows.Controls.CheckBox)(this.FindName("
                    checkLast")));
46              this.checkUpdate = ((System.Windows.Controls.CheckBox)(this.FindName
                    ("checkUpdate")));
47              this.buttonSave = ((System.Windows.Controls.Button)(this.FindName("
                    buttonSave")));
48          }
49      }
50  }
```

Listing C.10: Partial class of the page `Preferences`, obtained by Code-Generation

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Net;
5  using System.Windows;
6  using System.Windows.Controls;
7  using System.Windows.Documents;
8  using System.Windows.Input;
9  using System.Windows.Media;
10 using System.Windows.Media.Animation;
11 using System.Windows.Shapes;
12 using Microsoft.Phone.Controls;
13
14 namespace MapNavigator
15 {
16     public partial class Preferences : PhoneApplicationPage
17     {
18         public Preferences2()
19         {
20             InitializeComponent();
21         }
22
23         private void buttonSave_Click(object sender, RoutedEventArgs e)
24         {
25             NavigationService.Navigate(new Uri("/NavigateMap.xaml", UriKind.
                   Relative));
26         }
27
28         private void checkUpdate_Click(object sender, RoutedEventArgs e)
29         {
30
31         }
32
33         private void checkLast_Click(object sender, RoutedEventArgs e)
34         {
35
36         }
37     }
38 }
```

Bibliography

[1] Gartner. Gartner Says Sales of Mobile Devices in Second Quarter of 2011 Grew 16.5 Percent Year-on-Year; Smartphone Sales Grew 74 Percent. 2011.

[2] J.F. DiMarzio. Android - A Programmer's Guide. *McGraw-Hill*, 2008.

[3] M.Fowler. UML Distilled Third Edition. A Brief Guide to the Standard Object Modeling Language. *Addison Wesley*, 2006.

[4] Kendall Scott. UML explained. *Addison Wesley*, 2001.

[5] J.Warmer and A.Kleppe. The Object Constraint Language Second Edition, Precise Modeling with UML. *Addison Wesley*, 2003.

[6] M.Fowler. MDA explained. The Practice and Promise of the Model Driven Architecture. *Addison Wesley*, 2003.

[7] Object Management Group. UML 2.0 Specification. *OMG*, 2005.

[8] Microsoft Corporation. Xaml Object Mapping Specification. *MSDN Library*, 2008.

[9] Ayatullah Jibran Shidqie. Compilation of OCL into Java for the Eclipse OCL implementation. Master thesis, TU Hamburg-Harburg, 2007.

[10] Laszlo Lengyel, Tihamer Levendovszky, and Hassan Charaf. Implementing an OCL Compiler for.NET.

University of Verona
Department of Computer Science
Strada Le Grazie, 15
I-37134 Verona
Italy

http://www.di.univr.it