Model-Driven GUI Generation and Navigation for *Android* BIS Apps

Luís Pires da Silva[2] and Fernando Brito e Abreu[1,2]
[1]*DCTI, ISCTE-IUL, Av[a] das Forças Armadas, 1649-026, Lisboa, Portugal*
[2]*CITI, FCT/UNL, Campus da Caparica, Quinta da Torre, 2829-516, Caparica, Portugal*
luis.ptds@gmail.com, fba@{iscte-iul.pt, fct.unl.pt}

Keywords: Model-driven Generative Programming, Model-driven Navigation, *Android* GUIs, Usability in Mobile Apps.

Abstract: This paper presents our approach for producing graphical user interfaces (GUIs) for functionally rich business information system (BIS) prototypes, upon a mobile platform. Those prototypes are specified with annotated UML class diagrams. Navigation in the generated GUIs is allowed through the semantic links that match the associations and cardinalities among the conceptual domain entities, as expressed in the model. We start by reviewing the *Android* scaffolding for producing flexible GUIs for mobile devices. The latter can present rather different displays, in terms of size, orientation and resolution. Then we show how our model-based generative technique allows producing prototypes that match both the *Android* GUIs requirements, while implementing our model-driven approach for user navigation.

1 INTRODUCTION

The burst on the availability of smart phones based on the *Android* platform calls for cost-effective techniques to generate mobile apps for general purpose, cloud-based, business information systems (BIS). What drove us in doing this research was the need to find a better solution for the time consuming app creation problem, as recognized in (Parada and Brisolara, 2012).

To mitigate this problem our research aims at applying model-driven techniques to automatically generate usable prototypes with a sound, maintainable, architecture. Our generative approach is targeted to *Android* devices and produces GUIs that allow a conceptual navigation based on the relationships among domain entities (as described in a UML class diagram) and a few navigation genders. The GUI architecture can reach several screen sizes, resolutions, orientations and implements basic behavioural settings without any code repetition.

This paper is structured as follows: section two introduces the syntax used to describe the models and the proposed approach; section three introduces the *Android* and its characteristics and shows how we apply our approach to it; section four presents the tool that enables the solution; section five validates and compares our solution to related work; finally, in section six, we draw our conclusions and forecast future work.

2 GENERATIVE APPROACH

2.1 Model Specification

Our model specifications have two concrete syntaxes: graphical (UML class diagram) and textual. Annotations (functions starting with a "@") are only allowed in the textual representation. Let us consider the example shown in Figure 1 as our base model. The following example code shows the

```
@StartingPoint (NameToDisplay="Workers",
ImageToDisplay="")
@list (nickname="1")
@creation (nickname="1",salary="2")
@display (nickname="1",salary="2")
@unique (nickname="1",salary="2")
@domain()
class Worker
    attributes
        nickname: String
        salary: Integer
end
association Employs between
 Company[0..1] role employer
 Worker[1..*] role employees
end
```

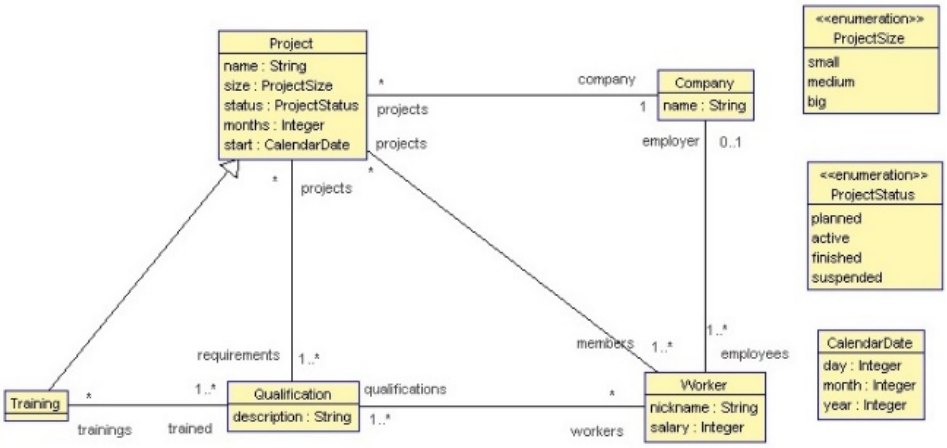

Figure 1: Projects World class diagram.

corresponding *Worker* class in textual format. Notice that the latter contains all the information deployed in the class diagram, plus the annotations, which are not depicted graphically. Annotations are used to describe functionalities and to assign purposes to specific attributes.

2.2 Prototype Navigation

Our navigation paradigm is based on a homomorphism between the traversal of the domain space and the GUI navigation space. We will now describe the three most important situations regarding GUI navigation: the starting point, navigation through associations and navigation aspects related to inheritance.

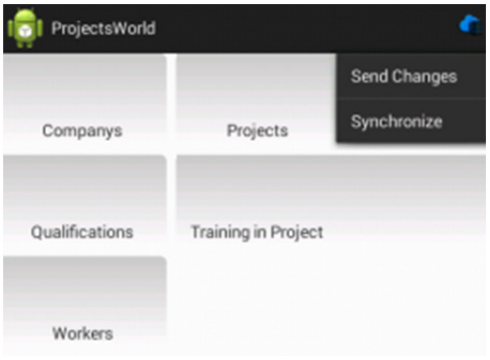

Figure 2: Projects World generated starting screen.

2.2.1 Starting Point

When the app is launched, the starting screen (Figure 2) presents the links to a set of preselected domain classes (using the *@StartingPoint* annotation). The rationale is granting a customizable entry point that will depend on the semantics of the app domain. For instance, it would not make sense to have in the entry screen the "many" side of a UML composition such as *ReceiptLine*, without going first to the *Receipt* class and selecting (or creating) its desired instance.

When the app user selects a domain entity (represented by a domain class annotated with @domain annotation) in the entry screen, the corresponding activity is rendered in one or two panes (e.g. Figure 3), depending on the device type. On the selected domain entity the app user can perform the usual CRUD operations or navigate to other related entities (by means of the top row navigation bar).

Figure 3: Training screen (2x one pane on smart phone).

Other present annotation will also fulfil a specific requirement namely: both @unique and @holder annotations are used for persistency purposes (they are explained elsewhere, since the topic of persistency is outside the scope of this paper); the @list annotation is used to specify which attributes will be shown in the list view (in this case only the *nickname* attribute will appear) and in what order; the @display annotation is used to specify the attributes that will be shown in the detail view, and

also their order; and lastly the @creation annotation is also used to specify which attributes are shown in an insert or update view, as well as their order.

Besides views generation, the previous specified attributes are also used to generate standard validation techniques (e.g., the view for an Integer attribute only accepts a natural number.

2.2.2 Navigating through Associations

Associations are the semantic pathways among domain entities and therefore we use them to navigate in the GUI. We have identified a limited set of domain traversal (navigation) genders and we assigned an icon to each one, as shown in Table 1. These icons are used in the navigation bar to provide semantic advice to the user, on where to move to.

Each domain traversal gender corresponds to a single movement from one domain entity to another, towards a UML association end (e.g. with cardinality one or many) or inheritance relation end (towards the parent or the children classes). For instance, in a hotel reservation app, while standing in the *Hotel* form, we would a get a "to many" icon for navigating to *Room* and a "to one" icon for navigating to *City*.

Table 1: Navigation icons.

Icon	Navigation Gender
	To one
	To many
	To associative
	To super
	To sub

2.2.3 Effect of Inheritance on Navigation

Each domain entity is directly coupled to a few others. This direct coupling can be seen as a semantic pathway and we use it for identifying the allowable navigations from one entity to the others.

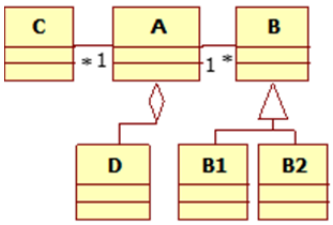

Figure 4: Navigation example.

In the example shown on Figure 4, the allowable navigations (target entities where the app user can navigate to), given the current context, are those represented in **Error! Not a valid bookmark self-reference.**.

Table 2: Navigation targets, for a given context.

Context	Navigation Targets
A	B, B1, B2, C, D
B	A, B1, B2
B1	B, A
B2	B, A
C	A
D	A

When the user navigates to B, we can have two situations: we get a *read-only* screen if B is abstract or a *read-write* one if B is not abstract.

2.2.4 Action based Navigation (Associating)

If instead of a normal click, the user does a long click over any choice in the navigation bar, the navigation process will still happen, but it will be for the creation of an association, i.e. the new screen will appear in *write-mode* (instead of the normal CRUD operations, only two options are available the *confirm button* and the *add button*) with all the selected entity instances, and from here the user can: (i) create a new instance that will automatically be associated with the previously selected instance; or (ii) associate to an already existing instance by selecting it and confirming the choice.

Finally, the *server button* with the cloud icon on the top-right corner in Figure 2 is used to synchronize data. If pressed, the user will be presented with two choices:

(i) *synchronize*, that allows synchronizing the local database with the one in the server;

(ii) *send changes*, that will send to the server every action performed in the session (since the app was launched or since the last "send.

3 MODEL-DRIVEN GUI GENERATION AND NAVIGATION

3.1 Android Architecture

Android follows a Model View Controller (MVC) architecture (Burbeck, 1987). The view is described in XML files. In the controller we have the so-called

"activities". One activity holds and inflates (renders) one or more XML files in one screen.

An *Android* application has many activities. Navigating means switching from one activity to another (i.e., from one screen to another). Activities are responsible for controlling every aspect of user interaction. This was the scenario until fragments were released, enabling the support for more dynamic and flexible GUI designs on larger screens.

A fragment is a modular section of an activity, which has its own lifecycle, can inflate XML files as well, receives its own input events, and can be added or removed while the activity is running. It also can be reused in different activities, thus providing a better support for a more organized application. Nevertheless, activities can still be considered the "screen", since they hold the fragments.

The native language support for *Android* apps is Java but only a few Java libraries are supported, as listed in the *Android* website (Google, 2013). Views are usually based on the aforesaid XML files, but they can also be made dynamic if written in Java. However, that is discouraged by Google, because XML-based views make the application run faster.

3.2 Screen Size and Orientation

There is a great diversity in screens available in mobile devices, regarding their resolution and size. For instance, Google (see *Table 3* in section *Supporting Multiple Screens* in (Google, 2013)) categorizes a set of prototypical screen sizes (*small screen, normal screen, large screen, extra-large screen*) and screen resolutions (*low density, medium density, high density, extra high density*). For instance, a *small screen* with *low density* can have a 240x320 resolution, while an *extra-large screen* with *extra high density* can have 2560x1600 pixels.

Besides the aforementioned combinations, we have the issue of screen orientation (portrait and landscape), which doubles the number of situations that an application needs to adapt to. In the absence of such an adaptation, apps will still work, but the views may suffer anamorphic distortion.

Summing up, it is very hard and time consuming to adapt mobile application GUIs to that variability. To reduce time-to-market, *Android* application developers often just provide support to the most common sizes, and disable screen rotation, a well-known fact for a frequent *Android* user.

3.3 Android Static Architecture

Android offers a set of default folders under the *res* folder (Figure 5) that hold configuration data required to automate screen rendering, depending on the mobile device being used, thus handling their different screen sizes and orientations. Since our goal is to generate these static user interfaces, by means of model-driven techniques, we have selected, as starting point, the default folders since they are a better fit to most configurations. Later, we plan to add the remaining options like *sw720dp*.

Inside the *res* folder there are five subfolders, each with a different purpose. The *layout* folder holds the XML files dedicated to the main interface description (i.e., the layout of the user interfaces and the components to be shown and their sizes).

The *menu* folder holds the XML files that describe the main menu view or in more recent *Android* versions the *ActionBar*.

The *values* folder holds the XML files that normally are used to store raw data. For instance, we can have a static list written in one XML as an array of values, and then bind (in Java) a list view with this array, thus granting extra flexibility (e.g. for

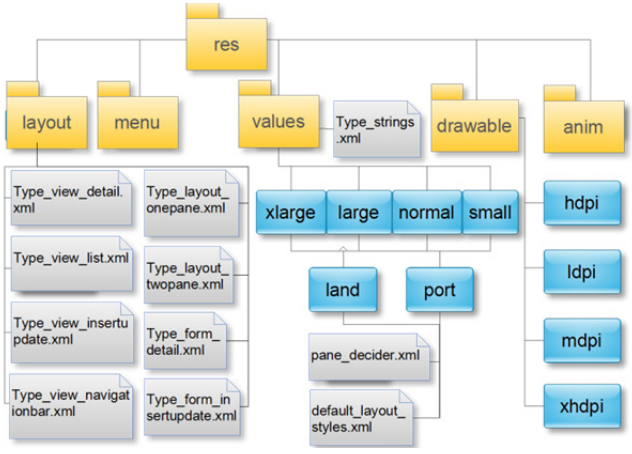

Figure 5: *Android* support for multiple layouts and resolutions.

customization or internationalization purposes).

The *drawable* folder holds any media file and/or XML files dedicated to drawing views. Finally, the *anim* folder holds XML files related to animations.

To illustrate how these folders are used in runtime, consider a mobile device with normal size. While in portrait orientation, the *values-normal-port* sub-folder is chosen but, if rotated to landscape, the *values-normal-land* sub-folder is chosen instead, if available. Otherwise, the default folder *values-normal* is selected. If the latter is missing, then the most basic folder (*values*) is selected.

A similar logic is applied to the other folders, although the *drawable* folder deserves further explanation, since this automatic reconfiguration capability relates to the screen resolution instead of size. For instance, if the mobile device has a low resolution screen, the *drawable-ldpi* folder is selected, while the *drawable-hdpi* is chosen if the screen is switched to high resolution. Therefore, it is normal in *Android* to see projects with media with the same content, but with different resolutions.

All mentioned folders and their contents (mostly XML files) are generated automatically in our generative, model-driven approach.

3.4 Different Layouts

Domain models are made of conceptual constructs understood by users, hereinafter called *domain entities*. For a hotel booking app we have entities such as *Room, Hotel, Reservation, Period* or *City*.

Domain entities are the first-class citizens of our generative approach. We create a form for each entity on the domain model (represented as a class type in a UML class diagram). For each form we consider two possible layouts: *two panes* for tablets and *one pane* for phones. A pair of XML files, one for each of the aforementioned layouts, is then generated for each domain entity. Their names are *Type_layout_onepane.xml* for one pane layouts and *Type_layout_twopane.xml* for two panes, where "Type" should be replaced by the name of the corresponding domain class, as for instance in *Receipt_layout_twopane.xml*.

We use a *master-detail flow* logic. On the *one pane* layout (phones), the default is showing the navigation bar and a list of objects. To get the detail of an object, the user makes a long-press click in it: the list of objects is hidden and the detail on the selected object is shown (e.g. as shown in Figure 3). On the *two pane* layout (tablets), both the list and the details of the currently selected object are shown simultaneously. This control logic is managed in the corresponding activity.

3.5 Presentation/ Navigation Logic

Data to be presented in forms, namely the view used for listing objects, is also defined by automatically generated XML files, one for each domain entity (as aforementioned): the *Type_view_list.xml* files control the representation of the object list views, the *Type_form* XMLs are merge-able XMLs – *Android* merge tag allows to create XMLs with the intent of being integrated in other XMLs, these files cannot used as views by themselves – and are responsible for the state representation (only attributes) of the object, while the *Type_view* XMLs files will hold the given forms, as shown in Figure 6.

The last two model-based generated XML files are the *Type_string.xml*, which contain raw data to be used by other XML files and, finally, the *Type_view_navigationbar.xml* files that show the navigation possibilities.

3.6 Enforcing Separation of Concerns

In a model-driven approach we work with an abstract representation of an object. This means that this object can present itself with different values. Therefore, we need to bind these values to the static GUI that we already introduced. To do so, while enforcing a good separation of concerns for

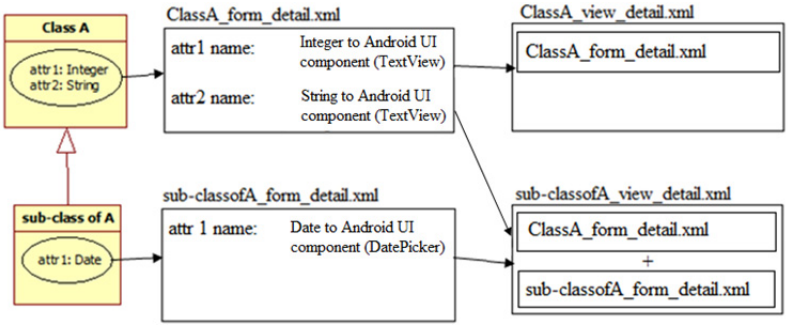

Figure 6: XML files generation and structure.

maintainability sake, a *fragment-based* approach was used, as advocated in the "*Building a dynamic GUI with fragments"* section of (Google, 2013). Thus the Model View View-Model (MVVM) was the chosen architecture. The MVVM is a more recent pattern, based on the MVC and developed by (Gossman, 2005) and based on the Presentation Model (Fowler, 2004), since both feature an abstraction of a View, which contains a View's state and behaviour, the difference is that Gossman presented the MVVM as a standardized way to leverage core features of the Windows Presentation Foundation (WPF) and Silverlight, in order to simplify the creation of user interfaces. Of course some adaptions had to be made in order to fully implement our solution in the *Android* platform, namely by passing every dynamic behavioural responsibility to the fragments classes (component listeners).

The proposed architecture is represented in Figure 7. Except for the *ListFragmentController,* every file presented is specific to one type of object, because the *ListFragmentController* represents a generic list with settable features, which can be settled by the main controller (the activity). So, regarding run-time code, we create two different fragments (box in Figure 7) for each domain model class, and a class that represents the object view in the list. The latter follows the *ViewHolder* pattern (Guy and Powell, 2010), since it improves performance significantly.

Since we have static ids in the correspondent components (the ones that we defined earlier), we

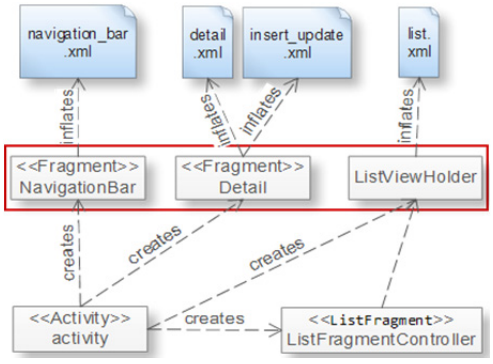

Figure 7: Proposed architecture: view and control layers.

use them when generating these classes. All the rest follows the template-based generation process.

Let us consider a TextView component to see the binding process. First, all the fragments must have two attributes defined: (i) the object that they workwith and (ii) the *rootView* of the type *View*, which represents all the views that the fragment works with. Then, each fragment will also have defined an instance of every component that it will need to show (due to the existence of dynamic data), but instead of creating an object, we bind with the proper static component by using the corresponding *id*. After binding the *rootView,* we have access to the components defined in it and we bind them as well.

The binding process must follow an adequate execution order. For that purpose, *Android* offers several methods. This is where a model-driven generation helps, since it properly adds the right code in the proper methods. If adjustments are required, the developer can do them without worrying with this type of side-effects since each fragment is responsible and can handle its own purpose, therefore increasing maintainability.

4 GENERATOR ARCHITECTURE

Our generative approach is implemented on top of *USE (Gogolla et al., 2007)*, an open-source tool developed at Bremen University. We have chosen this tool because it has a robust model compiler, supports model instantiation, annotations and, mainly, *OCL (Object Constraint Language)* for specifying model constraints (invariants, pre and post-conditions). Since USE is a standalone tool, with a GUI and other components of its own that we do not require for our model-driven approach, we have developed a façade component for USE named J-USE (Brito e Abreu, 2011) that allows accessing USE services conveniently.

We dubbed *JUSE4Android* our *Android* app generator that, from a domain model specified as

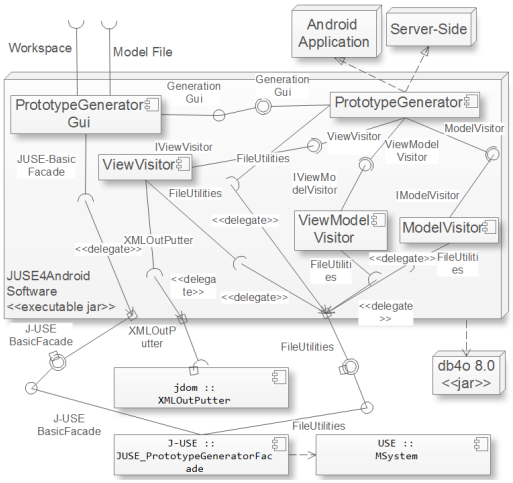

Figure 8: Architecture and Requirements of *USE4Android.*

described in the previous section, generates a full working prototype supporting the model-based navigation metaphor exploited in this paper.

Figure 8 presents the architecture of *JUSE4Android*. The visitor pattern was used in the code generator component. The link to the source code will be made available in the camera-ready version of this paper.

5 VALIDATION/ RELATED WORK

5.1 Related Work

Some related generative tools for Android exist. *Basic4Android* (Uziel) and *App Inventor* (MIT) follow a "visual programming" style. These are not model-driven tools and, in both cases, developing an app for a moderate sized domain would require a lot of effort. In our case, for an available model, the effort is very small, since it will mainly consist in defining a few annotations.

A closer related work, since it is also model-driven, can be found in (Parada and Brisolara, 2012). The input model is expressed through a UML class diagram and sequence diagrams, thus encompassing an increased modelling effort. The generation process itself requires advanced *Android* knowledge (i.e. the input for generation is not a plain PIM like in our case). A similar approach, suffering similar drawbacks, can be found in (Kraemer, 2011).

Last, but not the least, we have the *IBM Rational Rhapsody* (IBM) (D. Holstein, 2011) that presents itself as a complete model-driven solution for generating *Android* apps. However, in order to properly generate an app, every detail must be specified, making the code generation almost a mapping 1-to-1, thus encompassing a strong burden on the developer's side.

5.2 Case Study

We present herein our preliminary validation effort based on a case study – the *Projects World* project (Figure 1). Even for such a moderate small sized model, the output is considerably large, in both number of files and code length, even when applying the aforementioned code reuse techniques. In Table 3 is shown our tool full generative capabilities, i.e. besides outside the scope of this paper the tool also generates other layers with the exact same model. If this source code were produced manually, it would certainly corroborate the "time-consuming app creation problem" that we referred to in the introduction (Parada and Brisolara, 2012).

Table 3: Code generated for the *Projects World* exemple.

Layer	Type	Files	LOC
Business Layer(Model)	Java	17	4517
Control Layer (View-Model)	Java	4	420
Presentation Layer (View and View-Model)	XML Java	117 28	3345 8365
Persistency Layer	Java	1	230

It is worth mentioning that more than two thirds of the source code relates to the presentation layer, this is mainly due to the need of supporting a considerable range of screen sizes and resolutions for both orientations that characterize the multiple mobile devices that run *Android* nowadays. Without adequate code generation facilities like the one we presented herein, *Android* app programmers face "massive" code development.

Our goal to support different screens sizes and two different layouts (one pane for smart phones and two panes for tablets), both following the *Master-Detail Flow,* was met. Finally, by implementing proven techniques, namely by the usage of the *default_layout_styles.xml* and *pane_decider.xml* files separately, we provide an independent and feasible way to change resolutions, sizes and layouts to more specific goals outside the presented standard scope.

6 CONCLUSIONS & FUTURE WORK

On this paper we presented our GUI generation principles and navigability approach, aiming at producing BIS apps. We do not require the description of every possible scenario to generate a lot of screen sizes for both orientations and different devices running *Android*. We have shown how easily we can change one view for all possible configurations, based upon a UML class diagram and a template, thus avoiding "massive" code development.

Our MVVM-based architecture grants a separation of concerns that increases maintainability, namely by granting a *"strong separation between data, behavior, and presentation, making it easier to control the chaos that is software development"* (Smith, 2009).

We could not find any related work applying MVVM in the context of *Android.* Our architecture seems to be a good choice by comparison to other

mobile related implementations.

A set of problems are open for future research, namely the support for business rules and internationalization.

BIS applications require the definition of business rules. The latter can be as simple as setting a lower limit for marriage age or as complex as the preconditions for granting a bank loan or being refunded by the insurance in case of an auto collision. Thus, any BIS app generative approach will be incomplete unless that support is provided. At the model side we will use enrich our UML class diagrams with OCL clauses to specify the required BIS rules. Some interesting research problems then arise regarding where those rules will be verified (client or server) and how to grant state consistency on a distributed environment. Another issue will be generating automatic error dialogs with context-sensitive help.

Regarding internationalization, we basically need to provide support for different languages at the GUI side. As shown previously, we can change the layout in a single XML (corresponding to a domain entity) and that change will be propagated to all screen sizes and orientations. We intend to apply the same approach to the language support, since different languages will require different styles to adjust, namely in size, due to different average word lengths and desired verbosities (e.g. in contextual help).

We also plan to perform a systematic comparison of available generative approaches for *Android,* by using the same initial model as input and then assess the effect of requirements volatility. We are particularly concerned with the efforts required for:

(i) producing the input specification;
(ii) generating a baseline app;
(iii) adding extra requisites or removing existing requisites from the baseline app;
(iv) understanding the code of generated apps for maintenance sake.

Other aspects worth comparing include the usability of produced apps and their installability.

ACKNOWLEDGEMENTS

This work was partly supported by grant PEst-OE/EEI/UI0527/2011 of Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL).

REFERENCES

Brito e Abreu, F. 2011. *J-USE* [1.0]. Google Code: Google. Available at: https://code.google.com/p/j-use/ [Accessed: 12/10/2013].

Burbeck, S. 1987. Applications Programming in Smalltalk-80: How to Use Model-View-Controller MVC. Available at: http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html [Accessed: 12/10/2013].

D. Holstein, B. 2011. Speed Delivery of Android Devices and Applications with Model-Driven Development. Available at: http://www.ibm.com/developerworks/rational/library/model-driven-development-speed-delivery/model-driven-development-speed-delivery-pdf.pdf [Accessed: 12/10/2013].

Fowler, M. 2004. *Presentation Model*. Available at: http://martinfowler.com/eaaDev/PresentationModel.html [Accessed: 12/102013].

Gogolla, M., Buttner, F. & Richters, M. 2007. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming,* 69**,** pp. 27–34. Elsevier.

Google. 2013. *Android Developers*. Available at: http://developer.android.com/ [Accessed: 2013-01-07].

Gossman, J. 2005. *Model-View-ViewModel*. Available at: http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx [Accessed: 12/10/2013].

Guy, R. & Powell, A. 2010. *Google I/O 2010 - The World of ListView.* Google. Available at: http://www.youtube.com/watch?v=wDBM6wVEO70 [Accessed: 12/10/2013].

IBM. *Rational Rhapsody*. Available at: http://www-03.ibm.com/software/products/us/en/ratirhapfami [Accessed: 12/10/2013].

Kraemer, F. A. 2011. Engineering Android Applications Based on UML Activities. Proceedings of 14th International Conference on Model Driven Engineering Languages and Systems, pp. 183-197. Springer-Verlag.

MIT. *App Inventor*. Available at: http://appinventor.mit.edu/ [Accessed: 12/10/2013].

Parada, A. G. & Brisolara, L. B. d. 2012. A Model Driven Approach for Android Applications Development. *Brazilian Symposium on Computing System Engineering (SBESC'2012)*. Natal, Brazil.

Smith, J. 2009. WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN Magazine*. Available at: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

Uziel, E. *Basic4android*. Available at: http://www.basic4ppc.com/ [Accessed: 12/10/2013].