CrossMark

Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

FITEE

Generating native user interfaces for multiple devices by means of model transformation*

Ignacio MARIN[†1], Francisco ORTIN[†‡2], German PEDROSA[1], Javier RODRIGUEZ[1]

([1]*Department of Research and Development, CTIC Foundation, Gijon 33203, Spain*)

([2]*Department of Computer Science, University of Oviedo, Oviedo 33007, Spain*)

[†]E-mail: info@fundacionctic.org; ortin@uniovi.es

Received Mar. 19, 2015; Revision accepted Aug. 26, 2015; Crosschecked Nov. 11, 2015

Abstract: In the last years, the types of devices used to access information systems have notably increased using different operating systems, screen sizes, interaction mechanisms, and software features. This device fragmentation is an important issue to tackle when developing native mobile service front-end applications. To address this issue, we propose the generation of native user interfaces (UIs) by means of model transformations, following the model-based user interface (MBUI) paradigm. The resulting MBUI framework, called LIZARD, generates applications for multiple target platforms. LIZARD allows the definition of applications at a high level of abstraction, and applies model transformations to generate the target native UI considering the specific features of target platforms. The generated applications follow the UI design guidelines and the architectural and design patterns specified by the corresponding operating system manufacturer. The objective is not to generate generic applications following the lowest-common-denominator approach, but to follow the particular guidelines specified for each target device. We present an example application modeled in LIZARD, generating different UIs for Windows Phone and two types of Android devices (smartphones and tablets).

Key words: Model-to-model transformation, Native user interfaces, Model-based user interfaces, Model-driven engineering

doi:10.1631/FITEE.1500083 **Document code:** A **CLC number:** TP311

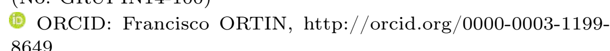
1 Introduction

Users access existing information systems using different types of devices such as desktop PCs, mobile phones, tablets, and embedded devices (present in vehicles, electrical appliances, and industrial applications). Their expectations include the possibility of using different native applications to access these information systems by using the various types of devices they own. This implies a challenge for developers, who must create their applications for many different kinds of platforms. Considering the amount of operating systems (OSs) and devices, plus their different versions, screen sizes, interaction mechanisms, and other hardware and software features, device fragmentation is one of the major problems that developers must deal with (Rajapakse, 2008).

Many research efforts have been carried out to propose techniques aimed at guiding the software engineering process to facilitate the creation of software for multiple devices (Berti *et al.*, 2004). Some of the research has been conducted in the scope of the model-based user interface (MBUI) generation paradigm. As stated in the W3C Model-Based

‡ Corresponding author

* Project supported by the European Commission's FP7 Serenoa Project (No. 258030), the National Program for Research, Development and Innovation, the Department of Science and Technology, Spain (No. TIN2011-25978), European Regional Development Funds (ERDF), European Union, and the Principality of Asturias, Science, Technology and Innovation Plan (No. GRUPIN14-100)


ORCID: Francisco ORTIN, http://orcid.org/0000-0003-1199-8649

©Zhejiang University and Springer-Verlag Berlin Heidelberg 2015

User Interface (UI) XG Final Report (W3C, 2010a), the purpose of model-based design is to identify high-level models that allow designers to specify and analyze interactive software applications from a more semantic-oriented level rather than addressing the implementation level. This allows the designers to focus on more important high-level aspects without being confused by implementation details. The implementation is defined later, using different software tools that follow the high-level models defined.

Existing MBUI tools generate generic, native (specific), and hybrid user interfaces. Those that generate generic UIs follow a lowest-common-denominator approach, abstracting the features that are supported by all the platforms and operating systems considered. SUIT is an example of this kind of MBUI (Pausch *et al.*, 1992). It distills the fundamental UI components provided by most platforms, which can be used by the SUIT programmer to implement Mac, Windows, and Unix UIs with identical look-and-feel (Pausch *et al.*, 1992). DIMAG is a device-independent mobile application generation framework that also follows the generic UI generation approach (Miravet *et al.*, 2014b). DIMAG translates UIs defined in IDEAL2 into code for Android, the MIDP Java ME platform, and the .Net Compact Framework (Miravet *et al.*, 2014a).

The main benefits of the generic UI approach are its simplicity, the easier generation process, and its commonly larger number of supported platforms. However, this approach does not exploit the full potential offered by a particular platform and its related interaction techniques. Therefore, the native UI generation approach is aimed at providing the user experience expected by software consumers. CIAT-GUI is an example of an MBUI method that supports the development of native usable UIs (Molina *et al.*, 2012). The CIAT-GUI tool enables the editing and validation of the UI models, the transformation between the intermediate representations, and the automatic generation of the final executable UI in extensible application markup language (XAML) (Molina *et al.*, 2012).

The hybrid generic and native UI systems combine both approaches. Gummy is an example of a hybrid generic and native UI approach (Meskens *et al.*, 2008). Gummy UI builder adapts its workspace to the particular elements of a specific target platform. Gummy builds a platform-independent representation, and updates it as the UI designer makes changes. With this platform-independent representation, Gummy generates an initial design for a new specific UI (Meskens *et al.*, 2008).

The existing MBUI works for multi-platform native application generation do not cover the transformation from concrete UI (i.e., a definition that depends on the type of target platform) to the final UI (i.e., the final definition of the application, either in source or in binary code) in depth. The missing point is that this transformation has been considered as a matter of translating 'concrete interactors' to the corresponding UI component in each target platform (W3C, 2010b). Therefore, the final look-and-feel is quite similar in the different final UIs generated. In addition, the final UIs use neither the different design guidelines defined by each OS manufacturer, nor the specific architectural patterns recommended (Section 8).

The main contribution of this work is a set of conceptual tools (meta-models) that provide a mechanism to guide the generation of final native UI applications from the concrete UIs, together with a software tool to support the proposed meta-models. Our proposal uses a simplified meta-model of the CAMELEON reference framework and the UsiXML language. Both the theoretical framework and the software tool have been named LIZARD, due to its similarity to CAMELEON, and considering that we envision it as a wizard to create multi-platform applications. LIZARD is freely available (Section 9).

Among all the types of cross-platform development tools, LIZARD is specialized in the generation of native mobile service front-end applications. 'Native' implies that the final application is compiled for a specific platform, as opposed to the Web approach (HTML application running in a Web browser) or to the hybrid approach (where most of the application view is achieved via an embedded Web browser). We have initially restricted applications to mobile platforms, because we think it implies sufficient platform heterogeneity to test our proposal. Finally, service front-ends are the main target in model-based UI generation, since the focus is the creation of the UI and the navigation model (business logic is defined aside).

2 LIZARD vs. CAMELEON

LIZARD is a modification of the CAMELEON framework (Calvary *et al.*, 2003; HIIS Laboratory, 2015) and the UsiXML language (Vanderdonckt *et al.*, 2004) that facilitates the generation of native service front-end applications, 'applications' in the rest of this paper, for different platforms and devices. The CAMELEON framework defines a set of meta-models hierarchized in terms of their abstraction level (Fig. 1), allowing the creation of context-aware applications. The definition of context includes software and hardware features of target devices in which the application will be executed.

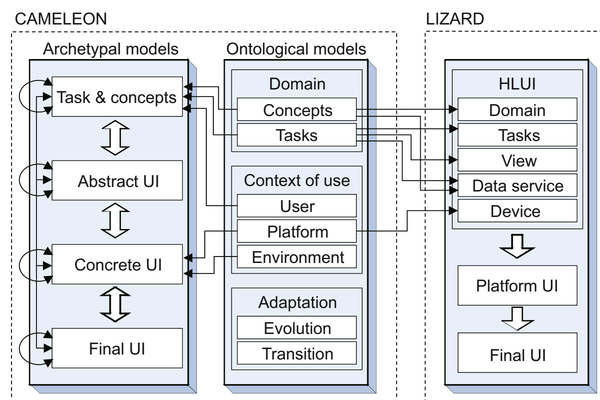


Fig. 1 Comparison between the CAMELEON and LIZARD frameworks

Fig. 1 shows how the CAMELEON reference framework recommends the specification of the domain, context of use, and adaptation ontological models (Calvary *et al.*, 2003). The domain model covers concepts (entities) and user tasks (activities). The context of use model is described in terms of the user, the platform, and the environment. The adaptation model specifies the reaction when context changes, comprising the evolution (the new UI to switch to), and the transition UI models.

The previous ontological models are instantiated into other archetypal models that serve as input to the design of a particular interactive system (left-hand side of Fig. 1). From the most abstract level to the most concrete one, the models produced in the design phases are tasks and concepts (TC), abstract UI (AUI), concrete UI (CUI), and final UI (FUI) (Calvary *et al.*, 2003). CAMELEON defines model transformations to adapt models from one abstraction level to another.

Our main objective is to enrich the CUI to FUI transformation to define an automatic generation of usable native UI applications. Thus, we have simplified the CAMELEON framework, grouping the TC and CUI meta-models abstraction levels—AUI is not modeled in LIZARD—into a single meta-model called the higher-level user interface (HLUI). Although CAMELEON indicates that there must be as many CUI languages as different interaction modes (e.g., vocal interaction and graphical interaction), in our simplified framework the CUI is limited to a single model for graphical interaction.

The HLUI abstraction level describes all the aspects required to model an application without mentioning the specific aspects of any platform or device. For instance, the HLUI does not consider the distribution of UI components in different screens—we use the term 'screen' to refer to the concrete screen of a concrete device. A typical problem in the context of this work is how to render the set of UI components of a view associated to a task, considering the dimensions of the mobile device screen.

HLUI includes the meta-models for application domains, tasks, views, data access, and devices, instantiating the concepts, tasks, and platform ontological models defined in CAMELEON (Fig. 1). The task model describes the tasks to be executed and their decomposition. Tasks are connected by transitions, and information may be passed from source tasks to the target ones.

The view model is also defined at the HLUI level. It is used to represent high-level layouts to render tasks, but it does not employ the concept of 'screen'. Layouts indicate how UI components (and other layouts) are grouped. Depending on the properties of the final device (such as the size of the screen and the structure of the layouts), the graphical components in a layout may be placed in the same view or in different ones. We have reused two ideas from Android and Windows Phone OSs to define the concept of 'view'. First, views resemble Android Fragments (Android Developers, 2015), allowing the definition of interface parts. These parts are used later, when tasks need to be rendered, and, depending on the target platform, the parts of a task may be represented in different screens or in the same one. The second idea is the inclusion of bindings, taken from the Windows Phone MVVM pattern (Smith, 2009). These bindings allow connecting the view widgets associated with a task to properties of the domain

model. Subsequently, the target views will manage the data input and output of the specific UI controls.

Once all the model instances are defined at the HLUI level, our framework transforms these models into instances of the new platform user interface (PUI) model (right-hand side of Fig. 1). One PUI model instance per target platform is generated. The PUI represents a new abstraction level in MBUI between the CUI and FUI levels, providing a concrete application model that considers the specific features of each target platform. It facilitates the generation of applications that follow the architectural and UI design patterns recommended in the guidelines of the different OS manufacturers. Table 1 summarizes the different patterns and guidelines that LIZARD uses for the target platforms and devices (detailed throughout the paper). For example, the concept of 'interface parts' is mapped to Android Fragments or Windows application pages; the generated code follows the MVP pattern for Android, and MVVM for Windows, and the representation of master/detail information in phones is split into two different screens, while one single master/detail screen is used for tablets.

Fig. 1 compares the CAMELEON and the LIZARD frameworks. Our framework is not an instantiation of CAMELEON because of three main dissimilarities. First, in LIZARD the developer models domain entities, tasks, views, and data services at the same level, and no translation is performed between these models. Second, LIZARD introduces a new PUI abstraction level between CUI and FUI, which represents the specific features of each target platform (the FUI level is equivalent in both approaches). Finally, LIZARD performs context adaptation depending only on the platform, without considering the user or environment ontological models defined in CAMELEON.

3 Architecture

We have developed a software tool that supports a model-driven process to generate native platform-specific applications. Its architecture is presented in Fig. 2, showing the different abstraction levels (HLUI, PUI, and FUI) and each module contained in the system. Fig. 2 also shows the technologies used to implement each module (dashed boxes), and their corresponding inputs and outputs (arrows).

The process to obtain the final application is linear, starting with the high-level description of the application at the HLUI level, and finishing with the generation of the final UI. The transformations from HLUI to PUI and from PUI to FUI are automatically done by the tool and require no manual tailoring. The FUI is the source code to be subsequently compiled by the corresponding compiler (e.g., the Android SDK or Visual Studio).

The proposed architecture follows a model-driven engineering (MDE) approach, implemented with different technologies included in the Eclipse Modeling Project (EMP) (The Eclipse Foundation, 2015b). We selected EMP because of its maturity and the easy integration with other model-based technologies.

3.1 Meta-modeling module

This module makes use of the Ecore meta-model tool in the Eclipse Modeling Framework (EMF, which is part of EMP) to define the LIZARD model instances shown in Fig. 2. EMF facilitates the definition of meta-models, the edition of model instances with its visual editor, and the integration with Xtext in order to define the Domain-Specific Languages (DSLs) for each meta-model. Although the defined models are based on UsiXML, we follow the abstraction level hierarchy defined in LIZARD (HLUI, PUI,

Table 1 UI style guidelines and patterns used for each target platform and device type

	Android phone	Android tablet	Windows Phone
Definition of 'interface parts'	Fragments		Application pages
Pattern used for the UI	Activity-based MVP		MVVM
Interaction with data	Explicit population of UI controls		XAML data bindings
Representation of master/detail information	2 different screens	1 master/detail screen	2 different screens
Decoupling data services from their implementation	Façade, Bridge and Factory Method design patterns		Funq dependency injection framework
Main navigation window	DashBoard	ActionBar	Pivot

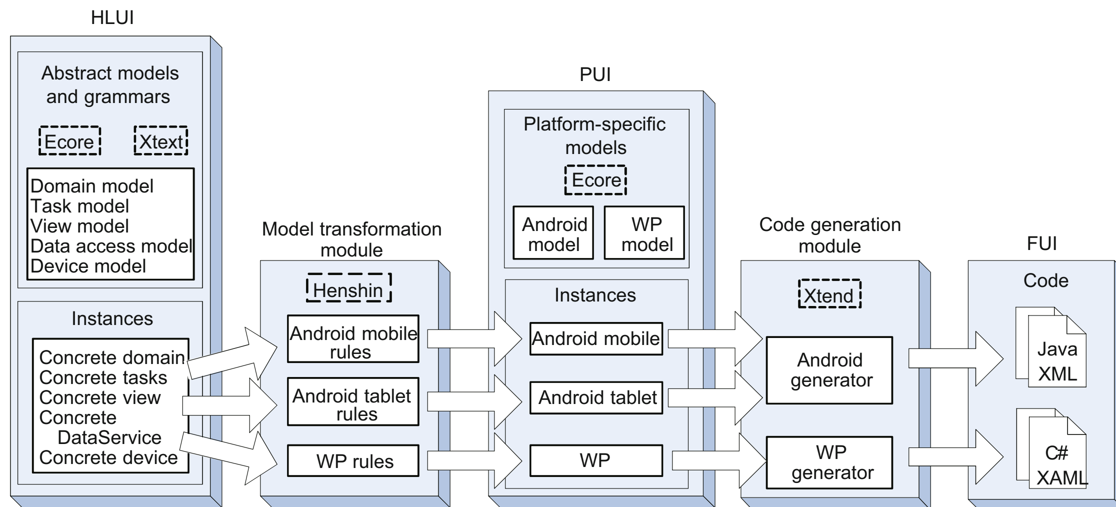


Fig. 2 Architectural diagram for the LIZARD software tool

and FUI) instead of the classical CAMELEON approach. As mentioned, the reason for this change is to generate platform-specific applications, restricting our approach to one single modality (i.e., graphical interaction).

The following subsections briefly describe each model. Section 4 analyzes each module by presenting an example application. Sections 3.1.1 to 3.1.5 are devoted to the models defined at the HLUI level in LIZARD, while Section 3.1.6 describes the PUI model. In addition to the Ecore models defined, equivalent Xtext grammars have been created to allow the description of applications with textual DSLs. As a result, the Eclipse IDE permits the edition of an instance model with both the Ecore visual model editor and a textual editor generated with Xtext.

3.1.1 Domain model

The domain model allows the description of the entities handled by the application, their properties, and the relationships among them. Its meta-model is presented in Fig. 3. A domain model is defined as a collection of DataTypes (e.g., String and int) and Entities. An entity may have one super-type, and consists of a collection of properties (Features). Example domain entities of the application presented in Section 4 are Book and Movie, while title and cover are two properties of these entities.

3.1.2 Task model

The task model is used to describe the actions that the user will be able to perform at runtime. Although it is based on the UsiXML task model, it additionally defines the execution flow of the application (also known as the navigation or dialog model). The execution flow defines the possible transitions between tasks, and the information passed in those transitions. Another feature introduced in our task model is the description of the data context handled by each task, i.e., the domain entities used by each task. For instance, the ‘view list of movies’ task (described in Section 4) is modeled expressing the data elements to be rendered and the flow to the ‘view movie detail’ task.

3.1.3 View model

The view model describes the visual appearance of the application, i.e., how each task in the task model is rendered. Fig. 4 shows an excerpt of the view meta-model. It defines a high abstraction model of layouts and widgets that can trigger events, executing actions of the registered listeners. At this level, the proposed system does not consider screens, because the distribution of UI components in each concrete screen is unknown at the HLUI level. Conceptually, this view model is equivalent to the UsiXML 1.8 concrete model for graphical modality (Limbourg and Vanderdonckt, 2004), which defines a set of widgets common to the majority of graphical interface toolkits. In our example, the

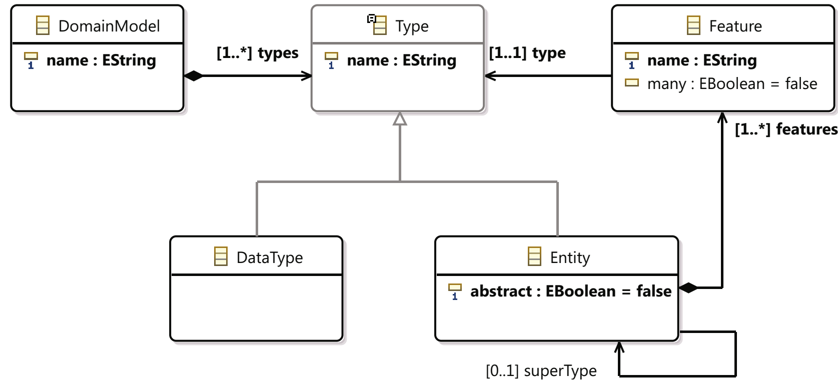


Fig. 3 Domain meta-model

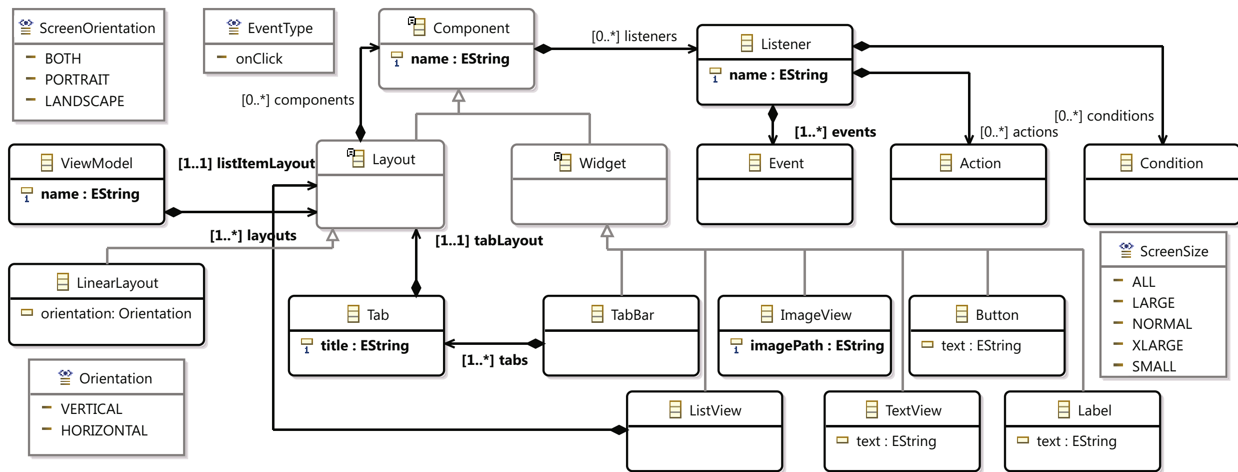


Fig. 4 Excerpt of the view meta-model

developer states that the application should render the 'view movie detail' task by showing the cover, title, director's name, or any other property of the corresponding movie. Additionally, it is possible to indicate the distribution of UI controls by means of layouts. Notice that this is not the final layout because the actual device in which the application will be executed is not known yet. Example layouts at this abstraction level may indicate that a control is located above another one, or that it is left-aligned (e.g., LinearLayout).

3.1.4 Data access model

The data access model describes the interface for obtaining the data handled by the graphical UI. This model decouples the application model from the underlying data access technology by means of dependency injection (Fowler, 2004). When an application has to display the existing list of movies, the information may be obtained from different sources (e.g., the device storage or a remote Web service). Therefore, the data access model represents only the data structure, leaving the data source as an implementation concern.

3.1.5 Device model

The device model defines the set of target devices for which a final application will be generated. For each device, its properties such as software platform (e.g., operating system, runtime environment, and version), device type (e.g., tablet or mobile phone), screen size, screen resolution, and screen depth are defined. These device properties are considered in the HLUI to PUI transformation and final code generation processes (Table 1). For instance, developers may indicate that they want the application to be generated for both Windows Phone smartphone and Android tablet. LIZARD will perform the

appropriate transformations to obtain the two PUI models, prior to generating the final code for both target devices.

3.1.6 PUI models

The PUI models allow the description of specific guidelines for each target platform. Instances of these models are generated by the system, applying transformations on the models described in the previous subsections (Fig. 2). PUI instances are the input of the code generation module. The PUI model includes the views for a particular platform, providing the activities (tasks) for the domain entities defined. Data is obtained from the data service implementation identified in the data access model.

In the PUI model, it is possible to define the recommended architectural and design patterns for a target platform. For instance, the platform-specific model for Windows Phone defines the MVVM architectural pattern, ensuring that this pattern is used following the particular application development guidelines (Table 1).

3.2 Instance creation module

EMF provides interesting features for the creation of instances of the models defined by LIZARD. Accordingly, developers may benefit from the use of the Eclipse IDE editors to create those instances. It also allows the use of the DSLs we have defined by means of Xtext grammars. A total integration with the Eclipse IDE is provided, assisting developers with features such as syntax coloring, quick fixes, content assist, autocomplete, and template proposals. Developers can create instances of the domain, task, view, data access, and device models by using the visual Ecore model editor or a textual DSL language. LIZARD uses these instances as the input for the model transformation module, generating instances of the PUI models for each device type indicated by the developer.

3.3 Model transformation module

The model transformation module takes instances of HLUI models and produces PUI model instances for each target platform. We have implemented this module using Henshin (Arendt *et al.*, 2010). Henshin is a declarative language to transform models based on graph transformation techniques, providing a graphical editor highly integrated with EMF. It enriches basic concepts from the domain of transformation rules with a powerful mechanism of expressing conditions and flexible attribute calculation. Henshin permits the definition of structures (transformation units) to control the application of rules in a modular way (Section 4.2). Henshin is implemented in Java and offers a strong integration with the rest of tools provided by EMP. Therefore, it makes it easy the collaboration with the rest of the modules described in Section 3.1.

This module generates model instances adapted to specific platforms and devices, translating generic models to more specific ones. The output instances employ specific UI controls and the architectural and design patterns of target platforms and devices. Section 4 illustrates some of the rules we have defined to perform this transformation.

3.4 Code generation module

The code generation module performs the last step to generate final applications. It explores the PUI model, which keeps references to the HLUI model, and reads the target platform specification. Afterwards, it generates the corresponding source code that will be compiled later. The availability of specific PUI models for each target platform facilitates the generation of code, close to that obtained by experienced developers.

We have implemented the code generation module in Xtend (The Eclipse Foundation, 2015d), a statically typed language integrated in the Java platform. Xtend is translated to Java source code, rather than to Java bytecode. Despite its general-purpose nature, its specific features make it especially adequate to implement code generators. Xtend facilitates the access to (and inspection of) Ecore model instances. Moreover, it offers template expressions with automatic handling of output code format. The multiple-dispatch mechanisms (aka multi-methods) are those that avoid the creation of complex visitors (Ortin *et al.*, 2007), extension methods, and lambda expressions.

4 Example application

4.1 Application description

We have implemented a prototype application to test the functionality of the different modules and the software tools. The example application is a program that allows users to query the details of various products grouped by categories. LIZARD creates a native application for each target device (an Android smartphone, an Android tablet, and a Windows Phone smartphone) from the very same application description. Each of the three target applications follows the set of architectural and UI design patterns, plus the guidelines defined for each platform and device type.

The example application allows users to select one out of three available categories: films, music, and books. Once a category has been selected, the user is provided with the list of elements in that category. Afterwards, it is possible to choose one of the elements to see its details, and perform an action depending on its media type (play an album track or open a URL of the IMDb film database).

From the UI point of view, the three generated applications vary according to different aspects (Table 1). For example, each application version offers a different way to select a category. The way the detail of each item in a category is presented is also different. Distinct options have been selected depending on the screen size, the UI components provided by the target platform, and the guidelines and recommendations for each platform (Table 1). The aim is to obtain an application as customized to the target device as possible. This will facilitate its utilization to the users familiar with the user experience of that platform.

As mentioned, main navigation (category choice) is realized in three different ways (Fig. 5), depending on the target platform and the device type (Table 1). The model transformation module (Section 3.3) is responsible for generating each of the patterns required, executing the appropriate rules:

1. Android tablet (Fig. 5, top): an ActionBar is rendered in the upper side of the screen, providing access to the different categories by means of tabs.

2. Android smartphone (Fig. 5, bottom left): a start screen with a button to access each category is rendered, following the DashBoard/Springboard design pattern (Neil, 2012).

3. Windows Phone smartphone (Fig. 5, bottom right): a Pivot control is used, showing related data in the same screen. It is possible to navigate from one view to another with left-to-right and right-to-left gestures.

LIZARD uses two alternatives to present master and detail information associated to the list of elements (Table 1):

1. Mobile phones, due to their reduced screen size, use independent screens for both master and detail views (Fig. 6). The initial screen offers the item list and, after selecting an item by clicking on it, the detail view is rendered.

2. For tablets, as they have larger screen sizes, the master/detail pattern is rendered in the same screen by using two panels (Fig. 7). The panel on the left shows the item list. The selection of one item updates the panel on the right, which shows the corresponding item details.

4.2 Application development

This section describes how the developer models the example application in LIZARD. The generation of the final application is automatically performed by our tool (Sections 5 and 6). The model instances are created with the Eclipse IDE, using the Ecore model editor or writing code in different DSLs. Fig. 8 shows the domain model of the example application in the Ecore model editor, whereas Fig. 9 displays the same application definition using the corresponding DSL.

Using one of these tools, the developer defines the following aspects of the application (detailed in the next subsections): the domain entities, the tasks that the user will be able to perform at runtime, the views and the high-level abstraction controls composing the graphical UI, the data services to populate UI controls, and the devices for which the application will be generated.

4.2.1 Domain entity description

In this model, the developer defines the domain entities, the definition of the attributes (including their data types and cardinalities) and the relationships among the entities. Figs. 8 and 9 show the entities defined in the example application (Product, Book, Film, and Album) and their attributes (title, image, director, etc.). Both figures also show other aspects supported by the domain meta-model,

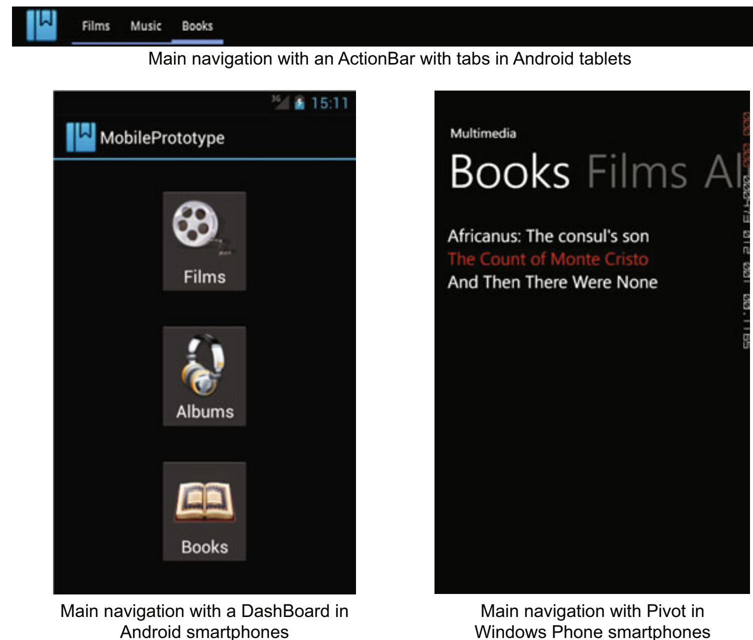


Fig. 5 Different navigation views depending on the target platform and device

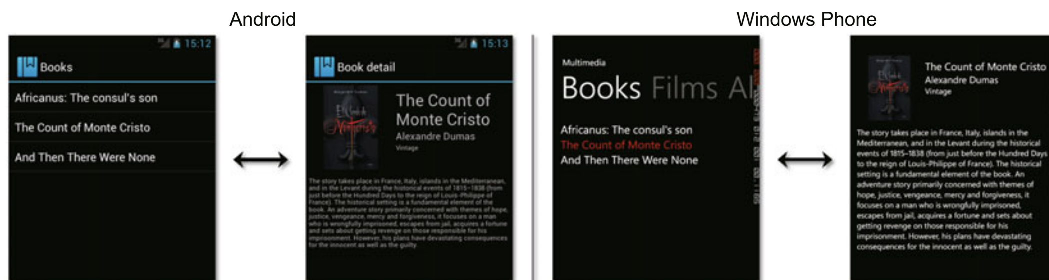


Fig. 6 Master-detail pattern on reduced screens

such as entity hierarchization. For instance, the entities Book, Film, and Album derive from the Product entity, which is declared as abstract.

4.2.2 Task description

At the task level, the developer defines the following aspects of the application (Fig. 10):

1. The set of tasks to be carried out by the application.

2. The relationships between tasks. It is possible to declare compound tasks that include several subtasks. The task to show music albums (musicTask) comprises two subtasks: one to show the list of albums (musicListTask), and the other to show the album details (musicDetailTask).

3. The data context handled by each task, based on relationships with the domain model. The task that shows the list of albums (musicListTask) must handle a list of Album entities (musicList, with one-to-many cardinality) and a property of type Album (selectedMusic) to keep the information related to the specific album selected by the user.

4. Transitions between tasks. There is a transition between the task that shows the list of albums and the task that displays the details of a specific album (toMusicDetail).

5. Information passed from the active task to the target task in a transition. The task showing the album list passes the selected album to the task showing the album details. Furthermore, the task model allows associating the data passed from the origin task to the destination task in a transition. For example, property musicTitle defined in the context of the target task (musicDetailTask) is mapped to

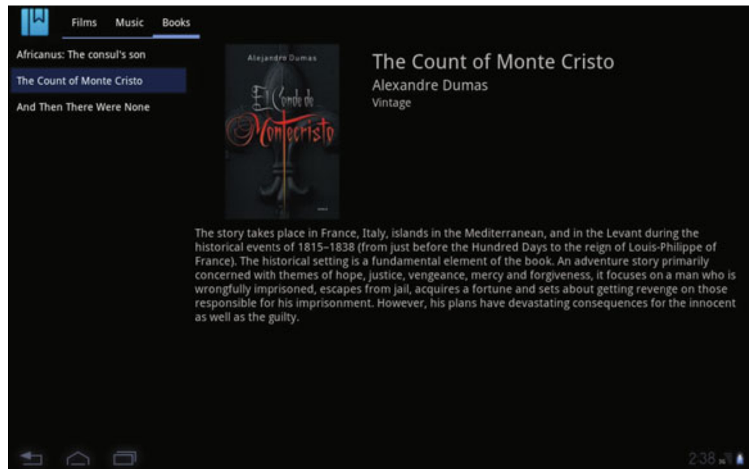


Fig. 7 Single screen master-detail presentation

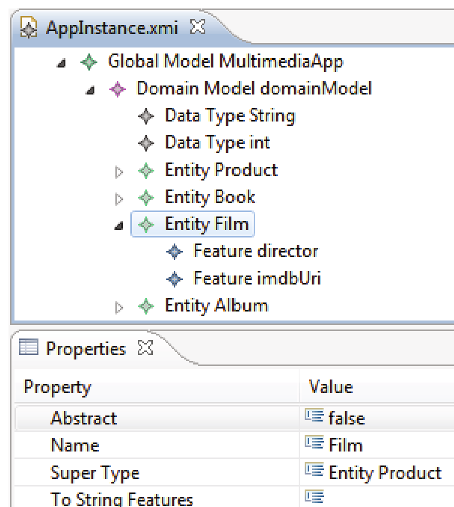


Fig. 8 Ecore model editor

the title attribute of the album selected by the user (property selectedMusic of the musicListTask active task)—the '^' operator refers to the Album object pointed by the selectedMusic property.

4.2.3 View description

The view model describes the abstract layout of each screen composing the UI of the application, and its components. It also allows the definition of the events triggered by the user. Fig. 11 shows a view model to render a list of music albums. The layout is defined by means of a single abstract ListView UI component.

The example in Fig. 11 includes several aspects to be considered. First, the task property of the layout indicates that the view is associated to the musicListTask task described in Fig. 10. This relationship enables the binding between visual controls (widgets) and the data context properties of the task. Accordingly, the musicListView control is able to use the selectedMusic property defined in the context of the associated musicListTask. Second, the layout of each widget may be defined within the ListView, a simple Label, in Fig. 11. Fig. 11 also shows that the property text of the Label is bound to the property title of a music album. Thus, the title of each

```
Domainmodel MultimediaDomain {
  datatype String
  datatype int

  abstract entity Product {
    String title
    String description
    String image
    toString : title
  }

  entity Book extends Product {
    String author
    String editorial
  }

  entity Film extends Product {
    String director
    String imdbUri
  }

  entity Album extends Product {
    int year
    String band
    String songUri
  }

}
```

Fig. 9 Domain model definition using a custom DSL

```
CompoundTask musicTask {
  title : "Music"
  relationshipType : independent

  Task musicListTask {
    title : "Albums"
    layouts : musicListLayout
    services : multimediaDataService

    DataContext musicListContext {
      Album [] musicList
      Album selectedMusic
      init musicList = getAlbums
    }

    Transition toMusicDetail {
      type : navigation
      target : musicDetailTask
      Param {
        contextProp : selectedMusic
        maps ^title -> musicTitle
        maps band -> musicBand
        maps description -> musicDescription
        maps image -> musicImage
        maps year -> musicYear
        maps songUri -> musicSongUri
      }
    }
  }

  Task musicDetailTask {
    title : "Album detail"
    layouts : musicDetailLayout
    DataContext musicDetailContext {
      String musicTitle
      String musicBand
      String musicDescription
      String musicImage
      int musicYear
      String musicSongUri
    }
  }
}
```

Fig. 10 Partial description of the task model (view of album list and album details)

music album in the list will be shown. Finally, another relevant aspect to be considered is the definition of Listeners that specify the actions to be carried out when an event occurs. Events are triggered by the actual UI components, when the user interacts with the view. The example in Fig. 11 declares a musicListItemListener for the event onClick associated to the Label. The action to be performed is the transition to the album detail view.

4.2.4 Data access description

This is the last model to be specified by the developer. This model defines the interface of the data access services required to populate the UI. The

```
LinearLayout musicListLayout {
  screenSizeSupport : ALL
  screenOrientationSupport : BOTH
  orientation : VERTICAL
  task : musicListTask

  ListView musicListView {
    itemsSource : musicList
    selectedItem : selectedMusic

    itemsLayout {
      Label musicListItemLabel {
        textSize : XXLARGE
        bind "text" to title
        Listener musicListItemListener {
          event : onClick
            Navigate(transitionRef:
                            toMusicDetail)
} } } } }
```

Fig. 11 List of music albums view definition

DataService model allows the declaration of each method used to obtain the data, indicating its name, parameters, and return type. In our example, it is necessary to declare the methods that provide the collections of books, music albums, and films from a data source (Fig. 12). The data will be used for the later population of the corresponding ListView widgets.

```
DataServiceModel {

  DataService multimediaDataService {

    op Book[] getBooks()
    op Film[] getFilms()
    op Album[] getAlbums()

    implementations:
      MultimediaDataServiceImpl,
      SQLMultimediaDataService

    defaultImpl:
      MultimediaDataServiceImpl
    }
}
```

Fig. 12 Data access definition

The data service model is strongly related to the domain model. In general, the data services provided are the typical create, retrieve, update, and delete operations against the entity objects defined in the entity model. In fact, the data to be received and returned by the data access services must be first defined in the entity model.

The DataService model decouples the data service from its implementation. A default implementation is used unless a specific platform model indicates the opposite. To avoid coupling between the

generated code and the concrete implementation to access the data, we have used the Façade, Bridge, and Factory Method (Gamma *et al.*, 1994) design patterns for Android, and the Funq (Clarius, 2015) Dependency Injection framework for Windows Phone (Table 1).

The implementation of the DataService represents the back-end of the application. It includes the business logic and the persistence layer. It can be local to each target device, using the local storage, or shared by all the target devices, using the same remote Web service. This is the only part of the final application that should be provided by the programmer; the rest is automatically generated by LIZARD from the HLUI models.

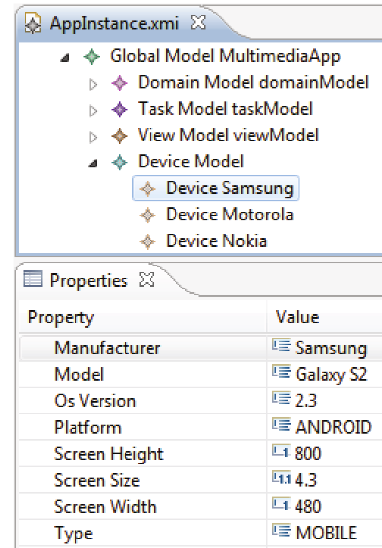


Fig. 13 Device properties definition

5 Model transformation

In the previous section we have described how the developer defines an application with high-abstraction models (HLUI). These models are common to all the target platforms and device types. In this step, the HLUI models are automatically transformed into a platform-specific (PUI) instance. One PUI model is obtained for each platform and device type.

Before doing that transformation, the target device and platform must be selected. Each device model expresses the set of properties associated to a target device, such as the screen size and resolution. Fig. 13 shows an example device model instance modeled with the Ecore model editor. These properties are used in both model transformation and code generation (Section 6). Currently, the transformation and code-generation rules use only the software platform (Android or Windows Phone) and the type of device (tablet or phone) properties. However, the device model also includes information about the manufacturer, model, screen size, screen width, screen height, and OS version for future uses.

5.1 Henshin

The software we have used to transform HLUI into PUI is Henshin, a model transformation language integrated with the various software modules developed in EMF. The transformation system is basically composed of two elements: rules modeled as graphs (EMF model instances) and transformation units. A rule is the most basic type of transformation, which is executed only once. A transformation unit is a control structure specifying the order in which rules are applied. They provide the sequential, iterative, and repetitive execution of rules or other transformation units. The developer indicates the transformation unit to be executed, which controls the sequence of rules that can be triggered. Therefore, the performance of graph pattern matching is improved because not all the rules need to be pattern-matched (Tichy *et al.*, 2013).

Rules are made up of a precondition graph (left-hand side, LHS), a postcondition graph (right-hand side, RHS) and, optionally, restrictions to rule application (application conditions). A transformation rule is applied on an instance of an EMF model, looking up the graph precondition in the model and substituting this graph by the postcondition graph, if the application conditions allow it.

Henshin provides a graphical notation that facilitates the definition, comprehension, and management of transformation rules. This graphical notation is based on colors and annotations (e.g., Fig. 14). The nodes of the LHS graph which persist after the execution of a rule (i.e., those that are in both the LHS and RHS) are presented in gray, with the annotation «preserve». The new nodes created in the RHS after the application of the rule are represented in green, with the annotation «create». Those nodes in the LHS to be deleted after the application of the rule are represented in red, with the annotation

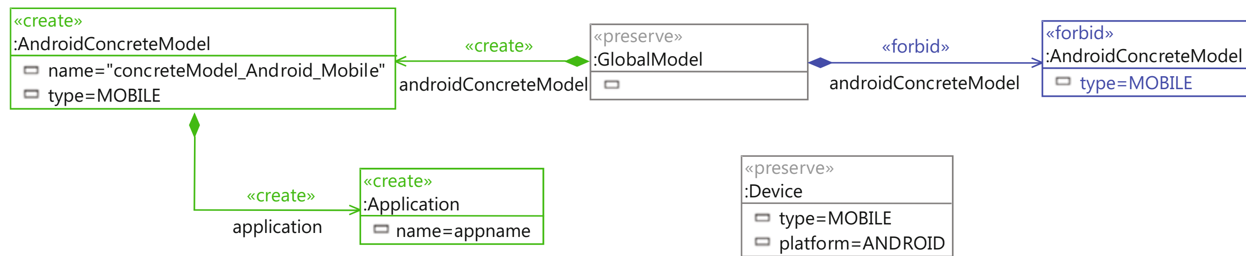


Fig. 14 Henshin rule to create a new PUI model (Android phone). References to color refer to the online version of this figure

«delete». Finally, application conditions are represented in Henshin with brown annotations «require», or with blue annotations «forbid», in the case of being a negative application condition (NAC). An NAC is a graph representing an additional condition: the graph to be transformed must not match the NAC graph.

In our example, we have implemented three sets of transformation rules: one for Android phones, one for Android tablets, and one for Windows Phone (WP). Table 2 (see p. 1012) shows the number of rules and transformation units used for each target platform. According to the taxonomy described by Mens and van Gorp (2006), the LIZARD rules can be classified as model, exogenous (source and target models are different), vertical (source and target models reside at different abstraction levels), equal technological space, and graph transformations.

These rules allow adapting applications to the different recommended software patterns of the distinct target platforms. As shown in Table 2, a different set of rules is applied to obtain a different PUI model for each target device. For each target platform and device-type combination, we have defined one different transformation aimed at defining the main navigation mechanism of the application: ActionBar with tabs for Android tablets, DashBoard for Android phones, and the Pivot control for Windows Phone. We have also specified transformations to provide the most suitable representation for master/detail information: a single screen with two panels for tablet devices, and two independent screens in the case of mobile phones.

For brevity, we detail only some of the rules summarized in Table 2. We first show some rules for Android smartphones (Figs. 14–18). Then, we present two more rules for Android tablets (Figs. 19 and 20) that offer alternative navigation via ActionBar UI controls.

5.2 Android phone

Fig. 14 shows the initial rule for the creation of a PUI model for Android mobile phones. As a precondition (LHS), it requires that the device model include a device with its type and platform properties set to MOBILE and ANDROID, respectively. If this precondition is satisfied, a new AndroidConcreteModel of type MOBILE containing a node of type Application is created. The NAC defined in that Henshin rule is used to prevent the rule from being executed more than once (because only one Android application for mobile phones should be generated). The execution of the rest of the rules defined for Android mobile phones completes this new model with all the nodes required for the subsequent code generation process.

Fig. 15 shows the rule for displaying master/detail information in two different screens. As it is devoted to the Android platform, a ListActivity is created to show the list of elements, and another Activity displays detailed information. As shown in the LHS (in gray), the precondition is satisfied if a CompoundTask composed of two subtasks exists, and there is a transition with information (Param) between both subtasks. It is also required that the first subtask include a ListView widget in its layout. The same result is obtained for the three categories (books, films, and albums) by invoking this rule in a loop transformation unit.

Figs. 16–18 show the rules and transformation units to create the DashBoard navigation control for Android mobile phones. The generation of patterns for the rest of devices is similar, obtaining a tabbed ActionBar or a Pivot control for Android tablets and WP smartphones, respectively. The execution of the rule in Fig. 16 creates a DashBoard for the Android-

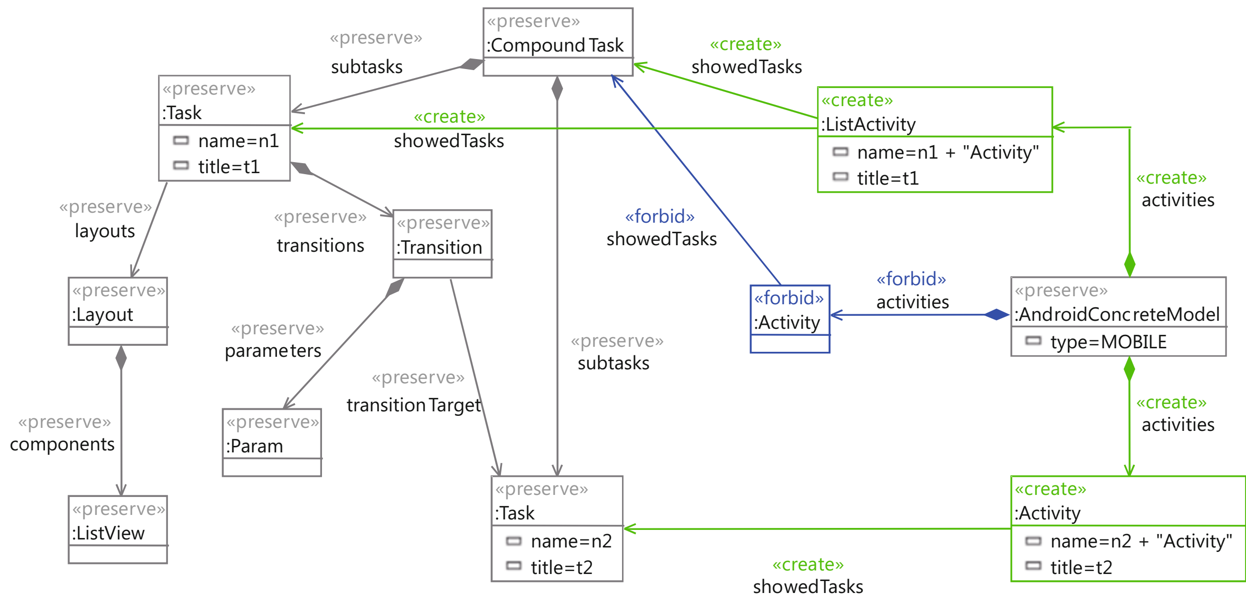


Fig. 15 Rule for visualization of master/detail information in two independent activities (Android phone). References to color refer to the online version of this figure

ConcreteModel in charge of providing the main task (initTask, in the task model) with a view. The values of some properties of the CompoundTask (e.g., title and mainTaskName) are used to initialize the properties of the new nodes (they are variables in Henshin).

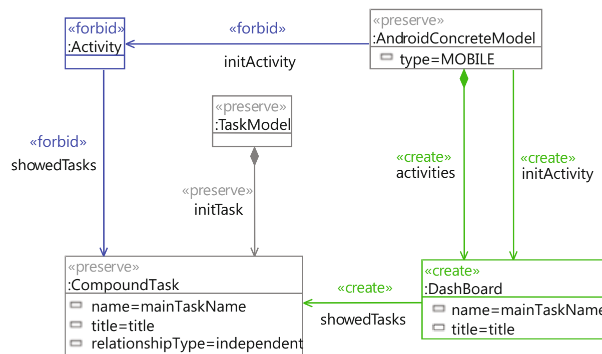


Fig. 16 Rule to create a DashBoard as main navigation mechanism (Android phone). References to color refer to the online version of this figure

The rule shown in Fig. 17 creates a DashBoardItem for each subtask of the main task. Each DashBoardItem refers to the activity associated to the task showing the list of elements in a category (books, albums, or films). Afterwards, in the code generation step, an Android layout XML file, an Activity class, and a Button class for each DashBoardItem are created, allowing the user to navigate in the list of elements of each category.

Fig. 18 shows the transformation units that control the execution of the rules previously shown for the creation of a DashBoard in Android mobile. The createNavigationWithDashBoard transformation unit triggers the sequential process that generates the navigation mechanism based on the DashBoard pattern. First, it invokes the rule in Fig. 16 (createMainActivityWithDashBoard) that generates the Dashboard. Afterwards, the createAllDashBoardElement loop transformation unit iteratively executes the rule in Fig. 17 (createDashBoardElement). Accordingly, a DashBoardItem for each subtask of the initial CompoundTask is created until a stop condition takes place—due to the NAC condition defined in the rule shown in Fig. 17.

5.3 Android tablet

In order to show the flexibility of our approach, we also present the transformation rules to obtain an Android tablet UI from the same HLUI model. Fig. 19 shows the Henshin rule used to create a navigation model based on a tabbed ActionBar. After the execution of this rule, an Activity with an ActionBar is created. This new Activity is associated to the main task (CompoundTask) by means of the showedTask reference. We ensure that the selected CompoundTask is the main task because of

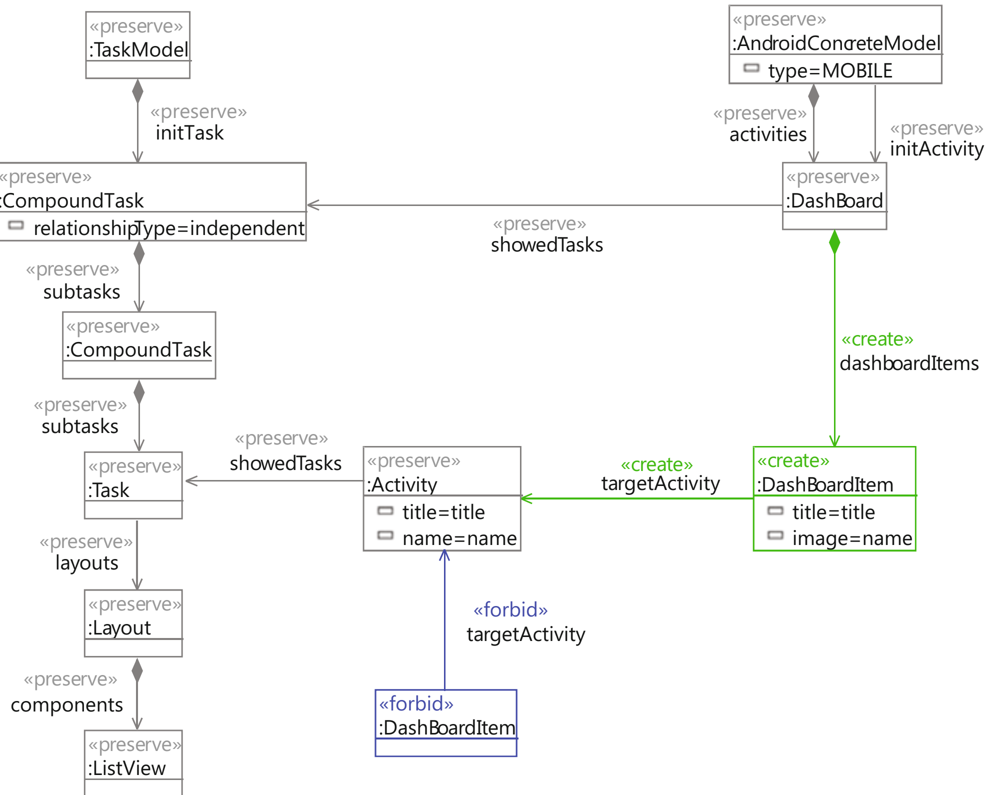


Fig. 17 Rule to create DashBoardItems for the navigation of each subtask in the main task (Android phone). References to color refer to the online version of this figure

the «preserve» condition (in gray), which indicates that the CompoundTask must be associated to the TaskModel by means of the initTask property. Subsequently, after the execution of the rule in Fig. 20 (invoked in a loop transformation unit), an ActionBarTab is created to access each child task, i.e., a tab per element category.

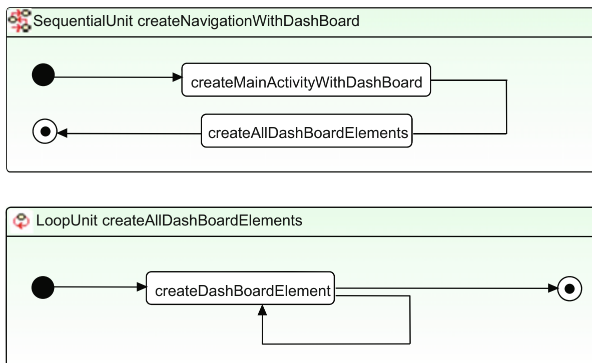


Fig. 18 Transformation units for execution control of the rules in Figs. 16 and 17 (Android phone)

The rules in Figs. 19 and 20 are an alternative to the navigation rules presented in Figs. 16 and 17. They show how Henshin facilitates the definition of different navigation patterns. For mobile devices, a DashBoard navigation view is generated (bottom-left corner of Fig. 5), whereas an ActionBar control is used for tablets (bottom-right corner of Fig. 5).

Note that Henshin rules are model instances. This fact makes it possible to create these rules at runtime. This feature may be used to extend the set of target platforms and devices at runtime, providing the generation of new type of applications when a specific platform/device is not supported.

6 Code generation

The code generation module receives a PUI model and generates the final application for the specific platform and device type. This process is also automatic, requiring no developer guidance. As mentioned, the code generation module has been implemented with the Xtend programming language, integrated in EMP.

Fig. 21 shows part of the Xtend class used to generate code for WP. The WPViewsCodeGenerator

Fig. 19 Henshin rule to create a main Activity, its layout, and an ActionBar (Android tablet). References to color refer to the online version of this figure

Fig. 20 Henshin rule to create a tab for each category (Android tablet). References to color refer to the online version of this figure

class implements the IGenerator interface, which forces the developer to implement a doGenerate method. This method receives two parameters. The first one, Resource, provides access to model instances, facilitating the iteration and filtering of the elements of the model. For example, line 9 in Fig. 21 illustrates how to filter ApplicationPage elements (the representation of a view or page in the WP PUI model) and iterate through them to generate C# and XAML code. The second one, IFileSystemAccess, offers disk access to produce the physical output. Output files are written with the generateFile method, passing the pathname of the output file as the first parameter, and the file content as the second one. In our example, the content is returned by the compileCodeBehind and compileXAML methods, generating C# and XAML code, respectively.

The generation of code behind (compileCodeBehind) is shown in lines 15 to 36 of Fig. 21. The two compileCodeBehind methods use the Xtend triple apostrophe feature that facilitates the use of code templates. All the code included between them is returned as a string except the code between '«' and '»', which is executed as common Xtend code. This feature facilitates the correct formatting of output code.

In its first line (line 16), the method creates a BasicNamespaceManager object to collect the required namespaces. In line 17, the overloaded version of the compileCodeBehind method (line 30) is invoked, storing the result in mainMethod. Afterwards, the added namespaces are iterated, writing them preceded by the 'using' reserved word. Finally, a namespace with the application name is defined,

```
01 class WPViewsCodeGenerator implements IGenerator {
02
03 @Inject extension WPGeneratorExtensions
04 GlobalModel gm
05
06 override doGenerate(Resource resource,
                       IFileSystemAccess fsa) {
07    gm = resource.allContents.filter(typeof(
                                 GlobalModel)).head
08
09    for (e : resource.allContents.toIterable.filter(
                          typeof(ApplicationPage))) {
10      fsa.generateFile(baseDirectory + 'View/' + e.name
          .toFirstUpper + ".xaml.cs", e.compileCodeBehind)
11      fsa.generateFile(baseDirectory + 'View/' + e.name
          .toFirstUpper + ".xaml", e.compileXAML)
12    }
13  }
14 }
15 def compileCodeBehind(ApplicationPage it) '''
16    «val usingManager = new BasicNamespaceManager()»
17    «val mainMethod = compileCodeBehind(usingManager)»
18
19    «IF !usingManager.getNamespaces.empty»
20       «FOR i : usingManager.getNamespaces»
21          using «i»;
22       «ENDFOR»
23    «ENDIF»
24
25    namespace «gm.appName».View {
26       «mainMethod»
27    }
28 '''
29
30 def compileCodeBehind(ApplicationPage it,
                         BasicNamespaceManager im) '''
31    public partial class «name.toFirstUpper» :
          «im.addNamespace("Microsoft.Phone.Controls")»
          PhoneApplicationPage {
32       public «name.toFirstUpper»() {
33          InitializeComponent();
34       }
35    }
36 '''
```

Fig. 21 Code generation for views in WP using Xtend

```
01 def compileXAML(ApplicationPage it) '''
02    «val namespaceManager = new BasicNamespaceManager()»
03    «val body = layout?.compileApplicationPageBody(
                                  namespaceManager)»
04    <phone:PhoneApplicationPage
05       «compileXAMLNamespaces(namespaceManager)»
06       «compileApplicationPageProperties»>
07
08       «body»
19
10    </phone:PhoneApplicationPage>
11
12 def dispatch compileApplicationPageBody(Layout it,
                          BasicNamespaceManager im) { }
13
14 def dispatch compileApplicationPageBody(
      LinearLayout it, BasicNamespaceManager im)'''
15    <Grid x:Name="LayoutRoot" Background="Transparent">
16       <StackPanel Orientation="«orientation.toString
            .toLowerCase.toFirstUpper»" Margin="8">
17          «components.map[compileComponent].join»
18       </StackPanel>
19    </Grid>
20 '''
```

Fig. 22 XAML code generation

including the declaration of a C# partial class that calls the inherited InitializeComponent method in its constructor. This code fraction shows how simple it is to use Xtend for code generation.

Fig. 22 shows part of the Xtend code required to generate the XAML code. Line 17 in Fig. 22 illustrates a relevant feature of Xtend: the use of the map extension function for lists. The 'map' is a high-order function that receives another function as a parameter, and invokes this function for each element of the list. It returns another list with the resulting values of invoking the function passed as a parameter. In our case, the compileComponent function (Fig. 23) is invoked for each element in the component list. Finally, all the results are combined in a String by means of the join function, and later included in an XAML StackPanel container.

Fig. 23 shows another important feature provided by Xtend. The compileComponent method is overloaded with different parameters. All of them use the dispatch reserved word in their definition. This makes all these methods become dynamically dispatched; i.e., they become multi-methods (Forax *et al.*, 2000). This means that the dynamic type of the parameter is used to resolve the method overload at runtime (Ortin *et al.*, 2014). In Java, multi-methods are commonly implemented with the Visitor design pattern (Gamma *et al.*, 1994). Therefore, the use of multi-methods simplifies code generation (Ortin, 2011). In our example, the single compileComponent XAML code template (line 7 in Fig. 23) is used for most of the widgets (TextBlock, TextBox, and Image). This template has been redefined only for specific widgets such as the ListView component (line 17), which includes the XAML elements required to define the items it contains.

Table 2 shows information about the generated applications for our example (in addition to the transformation rules discussed in the previous section). Except for the device model, the rest of HLUI

```
01 def dispatch compileComponent(LinearLayout it)'''
02    <StackPanel Orientation="«orientation.toString
          .toLowerCase.toFirstUpper»" Margin="8">
03       «components.map[compileComponent].join»
04    </StackPanel>
05 '''
06
07 def dispatch compileComponent(Widget it)'''
08    «IF listeners.empty || isWidgetIntoListView»
09       <«componentName» «compileWidgetProperties»/>
10    «ELSE»
11       <«componentName» «compileWidgetProperties»>
12          «compileAllListeners»
13       </«componentName»>
14    «ENDIF»
15 '''
16
17 def dispatch compileComponent(ListView it)'''
18    <ListBox «compileWidgetProperties»>
19       «compileAllListeners»
20          <ListBox.ItemTemplate>
21             <DataTemplate>
22                «listItemLayout.compileComponent»
23             </DataTemplate>
24          </ListBox.ItemTemplate>
25    </ListBox>
26 '''
```

Fig. 23 Xtend source code to generate the XAML components

models are exactly the same. The final application is made up of code that is generated by LIZARD, additional fixed code that is added to any application for a specific platform, and the code written by the programmer. In our example, this last kind of code is the implementation of a data service provider.

Apart from the use of Xtend, a key aspect that facilitates code generation is the utilization of specific PUI models for each platform and device type. Compared to CAMELEON, our proposed model is closer to target devices, resulting in a simpler and more maintainable development of the code generation module.

Table 2 Number of rules and transformation units used to transform HLUI into PUI, the lines of code (LOC) of each model specification, code generated, preexisting code, and code added by the programmer, and the number of existing files and files generated by LIZARD

Parameter	Android phone	Android tablet	Windows Phone
Number of rules applied	5	5	6
Number of transformation units applied	5	4	6
LOC (per model)			
Domain	27	27	27
Tasks	138	138	138
View	253	253	253
Data	10	10	10
Device	12	12	12
LOC of the generated code	1547	1694	4597
LOC of the additional code (fixed)	3375	2373	3856
LOC of the code added by the programmer	122	122	136
Number of files generated	53	62	60
Number of additional files	131	52	66

7 Evaluation

We evaluated the user satisfaction with the three different native UIs generated for our example. The objective of this evaluation is checking whether the final UIs generated by LIZARD are usable.

We used the Questionnaire for User Interaction Satisfaction, QUIS (Chin *et al.*, 1988). QUIS contains a demographic questionnaire, a measure of overall system satisfaction along six scales, and hierarchically organized measures of four specific interface factors (screen factors, terminology and system feedback, learning factors and system capabilities). Each area measures the overall satisfaction with each facet of the interface on a nine-point scale.

We defined three tasks to be done by the users with each target device: read the synopsis of a particular book, identify the director and year of a given film, and reproduce one album of a specific band. The application provides 50 different items for each category. The devices used were Nexus 5 (Android phone), Samsung Galaxy Tab 4 (Android tablet), and Nokia Lumia 930 (Windows Phone). Both the order of the tasks and devices, and the particular book, film, and album were chosen at random.

The three tasks were conducted by 15 users. Eight of them were expert software engineers, and 7 were common smartphone users; 11 were male and 4 female. The results of the demographic questionnaire revealed that 62% of the subjects fell in the range of 23–45 years old, and the average computer usage frequency was high.

After doing the three tasks with a particular device, the users were asked to answer the QUIS questionnaire. It measures the user satisfaction along

the following six scales: overall user reactions (S1), screen factors (S2), terminology and system information (S3), learning factors (S4), system capabilities (S5), and usability (S6) (Chin *et al.*, 1988). Each question has a scale from 1 to 9. An inter-item correlation analysis was carried out to validate the measures (Campbell and Fiske, 1959). An item is considered to be valid if its convergent validity (CV) is higher than its discriminant validity (DV). As a result, questions 12, 16, 30, and 31 were considered invalid, since their DV was higher than their CV.

Fig. 24 shows the mean values and the 95% confidence intervals for each scale. Similarly, Fig. 25 presents the distribution of the participant responses in a box plot. We can see in Fig. 24 how the average user satisfaction was at least 7.7 over 9 for all the devices and scales measured. Similarly, the minimum median was 8 (Fig. 25). Although the average user satisfaction for all the scales was higher when the tablet device was used (Fig. 24), no statistically significant difference existed (confidence intervals overlapped) (Georges *et al.*, 2007). Likewise, there was no significant difference between the Android and Windows Phone UIs for any scale.

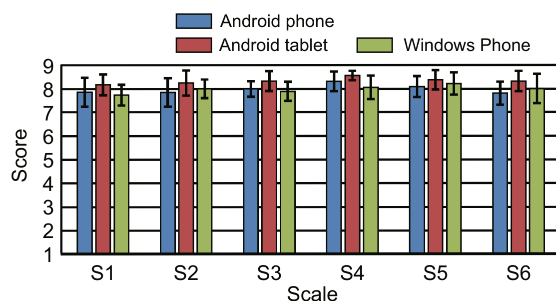


Fig. 24 Average responses and the 95% confidence intervals for each scale

LIZARD generates native UIs that follow the specific design guidelines defined by each OS manufacturer. This evaluation shows that the generated UIs provide good user satisfaction. Although LIZARD does not contribute to the user satisfaction by itself, its principle of generating different native UIs from the same model provides a high usability of the final applications.

8 Related work

CIAT-GUI is both a model-based UI development method and a software tool to generate final

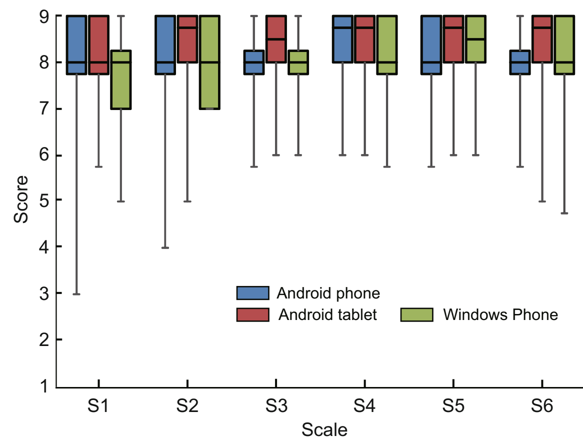


Fig. 25 Distribution of the participant responses for each scale

executable and usable UIs (Molina *et al.*, 2012). The development method begins with data (UML class diagrams) and tasks models (in Concur Task Trees, CTT, notation). Once the designer has created these two abstract models, he or she relates the tasks to be supported by the UI with the data handled by those tasks. This information serves as the input to a process that generates the AUI and CUI intermediate representations. The CUI is considered as an intermediate prototype that can be visualized, evaluated, and modified by end users. From the CUI, another process generates the final executable XAML code (Molina *et al.*, 2012).

MARIA is a method and the associated authoring tool for supporting the development of interactive applications able to access multiple Web services from different types of interactive devices (Paternò *et al.*, 2011). It addresses composition of Web services, both at the user interface and at the application level. MARIA considers some Web service annotations for the definition of UIs, using the 'ServFace' annotation meta-model. The tasks are first defined using CTT. Then, the operations in the Web services are associated to the tasks, generating AUI specifications expressed in MARIA, a model-based language for interactive applications (Paternò *et al.*, 2009). The abstract specifications are then refined into concrete, platform-dependent descriptions. Lastly, the implementations for the various target devices, which will support access to the Web services, are generated.

Ali (2004) proposed a framework for developing multi-platform UIs from a task model, supplemented with additional navigation attributes and

containment operators for each target platform. The task model, expressed in CTT, is transformed into a generic family model, which in turn is transformed into a platform-specific UI expressed in UIML (Ali *et al.*, 2002). The transformations are not automatic; they need developer guidance. The software tool generates Java Swing, Palm OS, and WML UIs.

UIML, the user interface markup language, is an XML language for declaratively defining UIs in a device-independent manner (Abrams *et al.*, 1999). The main objective is to permit a UIML document to be mapped to different types of UIs, from visual to speech. UIML represents an interface in five parts: interface structure, presentation style, content, actions taken in response to user interaction, and interconnection of the interface to application logic. UIML has been used in different MBUIs such as the system proposed by Ali (2004) and the Gummy frameworks (Meskens *et al.*, 2008).

Tran *et al.* (2009) defined a methodological process and a software tool to generate native UIs from task, domain, and user models. The task model is used to specify a generic UI; the domain model identifies the special features for creating the UI, and the user model is used to influence the design and to choose one solution in the design space. These three models are used to generate the native UIs for Java and .NET platforms, plus the basic functions of the database application.

The CAMELEON project is a relevant work in model-based UI design (Calvary *et al.*, 2003; HIIS Laboratory, 2015). It defines a conceptual framework based on a set of abstraction levels. CAMELEON is aimed at building methods and tools that support the design and development of highly usable context-sensitive systems. In the CAMELEON framework, the concept of 'context' refers not only to the execution platform, but also to the user preferences and interaction modes. The framework defines the transformations between the different levels, and proposes several languages and notations, such as UsiXML (Vanderdonckt *et al.*, 2004) and CTT, to express the meta-model of each abstraction level (Paternò *et al.*, 1997).

The most recent successor to CAMELEON is the Serenoa project (Caminero *et al.*, 2012). Serenoa has defined a Context-Aware Reference Framework (CARF) that specifies the relevant concepts to implement and perform context-aware adaptation (Serenoa, 2012). It comprises seven branches that contain potential instances for implementing, performing, and analyzing context-aware adaptation: how, to what, where, why, what, who, and when to apply adaptation. The 'to what' branch considers the user, the platform, and the environment. LIZARD is focused on the platform instance of this branch.

Degrandsart *et al.* (2014) proposed a context-aware application model in which context adaptations were specified as model transformations, generating the final application for Android phones. The UI was identified as a relevant aspect in the context of a platform-independent model, together with structure (UML class diagrams) and activity (UML activity diagrams) models. Platform-independent UIs were specified in a special purpose modeling language called CAP3 (van den Bergh *et al.*, 2011). CAP3 models abstract UIs by integrating structural and behavioral specifications. CAP3 includes explicit references to domain, user, and context models (van den Bergh *et al.*, 2011).

Gummy is a multi-platform graphical UI builder (Meskens *et al.*, 2008). It generates an initial general design for a new platform, adapting and combining the features of existing platform-specific UIs created for the same application. A solution for designing a UI for different platforms is by transforming high-level models into platform-specific UIs. However, the resulting UIs commonly lack the aesthetic quality of a manually designed UI. Gummy combines the benefits of both the manual approach and model-based techniques. It builds a platform-independent representation of the UI expressed in UIML, and updates it while the designer makes changes. Gummy provides this abstract UI representation as the initial design for a new platform.

The MBUI paradigm has also been used in other scenarios such as the development of sustainable visualization solutions in industrial automation. An example is Movisa, a DSL designed to capture the functional contents of visualization solutions, where the specific technical realization elements are added by means of model transformations (Hennig and Braune, 2011). Movisa defines the presentation (UI), algorithm (application-specific logic), and client data (data whose values affect the UI) models (Cabot, 2013). The meta-models are specified with Eclipse EMF (The Eclipse Foundation, 2015a), and model

transformations are defined with Epsilon EGL (The Eclipse Foundation, 2015c). These transformations generate Web and Java UIs, using the Model-View-Presenter and Model-View-Controller patterns.

Aquino *et al.* (2010) performed a study to evaluate the usability of multi-device and multi-platform UIs generated with an MDE approach. Their conclusion was that the tested MDE approach should incorporate enhancements in its multi-device and multi-platform UI generation process, to improve the usability of the generated applications. The main cause of this conclusion is that the UI generation process is commonly limited to translating the same user experience across all the platforms. Applications generated for different platforms should consider the specific guidelines defined for the target environment (Aquino *et al.*, 2010).

Responsive web design is an attempt to optimize the viewing experience across different devices, adjusting for distinct viewport sizes, resolutions, and contexts (Marcotte, 2011). A site designed with responsive design adapts the layout to the viewing environment by using fluid grids, flexible images, and media queries to use different styles based on the characteristics of the particular device (W3C, 2012). The adaptation of the UI is performed dynamically, depending on the device used, whereas the MBUI approach generates a specific UI statically. Besides, MBUI can be used to generate web, native, and hybrid UIs.

Existing approaches for transforming CUI to FUI are focused on the use of the Visitor design pattern and template-based approaches (Limbourg and Vanderdonckt, 2009). The main disadvantage of the Visitor design pattern is that it does not solve the expression problem (Cook, 1991), produced when recursively defined data types and their operations have to be extended simultaneously (Ortin and Garcia, 2011). The use of multi-methods has been previously identified as a suitable approach for overcoming this limitation (Zenger and Odersky, 2005).

9 Conclusions

A set of conceptual HLUI meta-models representing the domain entities, tasks, high-level application views, and data services have been proposed to generate final native UI applications, which follow the recommended guidelines for each target platform. The developer defines these high-level models and identifies the selected target device. Following the MBUI approach, the HLUI models are transformed into a platform-specific model (PUI) representing the target application. A code generation process produces the final application from the PUI model. The creation of model instances, the HLUI to PUI transformation, and the code generation process are supported by a software tool. Both the conceptual framework and the software tool are called LIZARD.

An example application has been created in LIZARD. After modeling the application, the final UI is automatically generated by the software tool. The developer has to provide an implementation of the data service module before running the application. We support Windows Phone and two types of Android devices (phones and tablets). An evaluation has shown that the average user satisfaction with the three generated applications is at least 7.7 over 9 for all the scales and devices measured. The generation of different native UIs considering the guidelines defined for each target platform and device has confirmed the expected user satisfaction.

Our example application is a proof of concept to illustrate the feasibility of our proposal. More sophisticated applications should be defined to validate the appropriateness of LIZARD for real applications. We plan to extend LIZARD to support other mobile target platforms such as iOS, and desktop systems such as Windows, Mac OS, and Linux. We also intend to enrich the LIZARD device models with a device description repository (DDR).

The LIZARD prototype, its source code, and the example application presented in this paper are freely available at https://bitbucket.org/fundacionctic/lizard.

References

Abrams, M., Phanouriou, C., Batongbacal, A.L., *et al.*, 1999. UIML: an appliance-independent XML user interface language. *Comput. Networks*, **31**(11-16):1695-1708. [doi:10.1016/S1389-1286(99)00044-4]

Ali, M.F., 2004. A Transformation-Based Approach to Building Multi-platform User Interfaces Using a Task Model and the User Interface Markup Language. Faculty of the Virginia Polytechnic Institute and State University.

Ali, M.F., Perez-Quiñones, M.A., Abrams, M., *et al.*, 2002. Building multi-platform user interfaces with UIML. Computer-Aided Design of User Interfaces III, p.255-266. [doi:10.1007/978-94-010-0421-3_22]

Android Developers, 2015. Fragments Developers. The Android Fragments API. Available from http://developer.android.com/guide/components/fragments.html

Aquino, N., Vanderdonckt, J., Condori-Fernández, N., *et al.*, 2010. Usability evaluation of multi-device/platform user interfaces generated by model-driven engineering. Proc. ACM-IEEE Int. Symp. on Empirical Software Engineering and Measurement, p.30.1-30.10. [doi:10.1145/1852786.1852826]

Arendt, T., Biermann, E., Jurack, S., *et al.*, 2010. Henshin: advanced concepts and tools for in-place EMF model transformations. Proc. 13th Int. Conf. on Model Driven Engineering Languages and Systems: Part I, p.121-135. [doi:10.1007/978-3-642-16145-2_9]

Berti, S., Correani, F., Mori, G., *et al.*, 2004. TERESA: a transformation-based environment for designing and developing multi-device interfaces. Proc. CHI Extended Abstracts on Human Factors in Computing Systems, p.793-794. [doi:10.1145/985921.985939]

Cabot, J., 2013. Movisa: a DSL Tool for Human Machine Interfaces (HMI) in Industrial Automation. Available from http://modeling-languages.com/movisa-a-dsl-tool-for-human-machine-interfaces-hmi-in-industrial-automation

Calvary, G., Coutaz, J., Thevenin, D., *et al.*, 2003. A unifying reference framework for multi-target user interfaces. *Interact. Comput.*, **15**(3):289-308. [doi:10.1016/S0953-5438(03)00010-9]

Caminero, J., Rodríguez, M.C., Vanderdonckt, J., *et al.*, 2012. The SERENOA project: multidimensional context-aware adaptation of service front-ends. Proc. 8th Int. Conf. on Language Resources and Evaluation, p.2977-2984.

Campbell, D.T., Fiske, D.W., 1959. Convergent and discriminant validation by the multitrait-multimethod matrix. *Psychol. Bull.*, **56**(2):81-105. [doi:10.1037/h0046016]

Chin, J.P., Diehl, V.A., Norman, K.L., 1988. Development of an instrument measuring user satisfaction of the human-computer interface. Proc. SIGCHI Conf. on Human Factors in Computing Systems, p.213-218. [doi:10.1145/57167.57203]

Clarius, 2015. Funq: a Fast DI Container You Can Understand. Available from http://funq.codeplex.com

Cook, W.R., 1991. Object-oriented programming versus abstract data types. Proc. REX School/Workshop on Foundations of Object-Oriented Languages, p.151-178. [doi:10.1007/BFb0019443]

Degrandsart, S., Demeyer, S., van den Bergh, J., *et al.*, 2014. A transformation-based approach to context-aware modelling. *Softw. Syst. Model.*, **13**(1):191-208. [doi:10.1007/s10270-012-0239-y]

Forax, R., Duris, E., Roussel, G., 2000. Java multi-method framework. Proc. Int. Conf. on Technology of Object-Oriented Languages and Systems, p.45-56.

Fowler, M., 2004. Inversion of Control Containers and the Dependency Injection Pattern. Available from http://martinfowler.com/articles/injection.html

Gamma, E., Helm, R., Johnson, R., *et al.*, 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Boston, MA, USA.

Georges, A., Buytaert, D., Eeckhout, L., 2007. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Not.*, **42**(10):57-76. [doi:10.1145/1297105.1297033]

Hennig, S., Braune, A., 2011. Sustainable visualization solutions in industrial automation with Movisa—a case study. IEEE Int. Conf. on Industrial Informatics, p.634-639.

HIIS Laboratory, 2015. The CAMELEON Reference Framework. Plasticity of User Interfaces. Available from http://giove.isti.cnr.it/projects/cameleon.html

Limbourg, Q., Vanderdonckt, J., 2004. UsiXML: a user interface description language supporting multiple levels of independence. Engineering Advanced Web Applications: Proc. Workshops in Connection with the 4th Int. Conf. on Web Engineering, p.325-338.

Limbourg, Q., Vanderdonckt, J., 2009. Multi-path transformational development of user interfaces with graph transformations. Human-Centered Software Engineering—Software Engineering Models, Patterns and Architectures for HCI, p.107-138. [doi:10.1007/978-1-84800-907-3_6]

Marcotte, E., 2011. Responsive Web Design. Jeffrey Zeldman, New York, NY, USA.

Mens, T., van Gorp, P., 2006. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, **152**:125-142. [doi:10.1016/j.entcs.2005.10.021]

Meskens, J., Vermeulen, J., Luyten, K., *et al.*, 2008. Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me. Proc. Working Conf. on Advanced Visual Interfaces, p.233-240. [doi:10.1145/1385569.1385607]

Miravet, P., Marin, I., Ortin, F., *et al.*, 2014a. Framework for the declarative implementation of native mobile applications. *IET Softw.*, **8**(1):19-32. [doi:10.1049/iet-sen.2012.0194]

Miravet, P., Ortin, F., Marin, I., *et al.*, 2014b. Using standards to build the DIMAG connected mobile applications framework. *Comput. Stand. Interf.*, **36**(2):354-367. [doi:10.1016/j.csi.2013.08.007]

Molina, A.I., Giraldo, W.J., Gallardo, J., *et al.*, 2012. CIAT-GUI: a MDE-compliant environment for developing graphical user interfaces of information systems. *Adv. Eng. Softw.*, **52**:10-29. [doi:10.1016/j.advengsoft.2012.06.002]

Neil, T., 2012. Mobile Design Pattern Gallery. Addison-Wesley, Sebastopol, CA, USA.

Ortin, F., 2011. Type inference to optimize a hybrid statically and dynamically typed language. *Comput. J.*, **54**(11):1901-1924. [doi:10.1093/comjnl/bxr067]

Ortin, F., Garcia, M., 2011. A type safe design to allow the separation of different responsibilities into parallel hierarchies. Evaluation of Novel Approaches to Software Engineering, p.15-25.

Ortin, F., Zapico, D., Cueva, J.M., 2007. Design patterns for teaching type checking in a compiler construction course. *IEEE Trans. Educ.*, **50**(3):273-283. [doi:10.1109/TE.2007.901983]

Ortin, F., Quiroga, J., Redondo, J.M., *et al.*, 2014. Attaining multiple dispatch in widespread object-oriented languages. *DYNA*, **81**(186):242-250. [doi:10.15446/dyna.v81n186.40428]

Paternò, F., Mancini, C., Meniconi, S., 1997. ConcurTaskTrees: a diagrammatic notation for specifying task models. Proc. IFIP TC13 Int. Conf. on Human-Computer Interaction, p.362-369.

Paternò, F., Santoro, C., Spano, L.D., 2009. MARIA: a universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Human Interact.*, **16**(4):19.1-19.30. [doi:10.1145/1614390.1614394]

Paternò, F., Santoro, C., Spano, L.D., 2011. Engineering the authoring of usable service front ends. *J. Syst. Softw.*, **84**(10):1806-1822. [doi:10.1016/j.jss.2011.05.025]

Pausch, R., Conway, M., Deline, R., 1992. Lessons learned from SUIT, the simple user interface toolkit. *ACM Trans. Inform. Syst.*, **10**(4):320-344. [doi:10.1145/146486.146489]

Rajapakse, D.C., 2008. Fragmentation of Mobile Applications. *In*: Alencar, P., Cowan, D. (Eds.), Handbook of Research on Mobile Software Engineering. Engineering Science Reference, Hershey, PA, USA.

Serenoa, 2012. Multi-dimensional Context-Aware Adaptation of Service Front-Ends. Deliverable 2.1.2 CARF and CADS (R2).

Smith, J., 2009. WPF Apps with the Model-View-ViewModel Design Pattern. *MSDN Mag.*, **2**(1):1-19.

The Eclipse Foundation, 2015a. EMF, the Eclipse Modeling Framework. Available from https://www.eclipse.org/modeling/emf

The Eclipse Foundation, 2015b. EMP, the Eclipse Modeling Project. Available from http://www.eclipse.org/modeling

The Eclipse Foundation, 2015c. Epsilon Generation Language. Available from http://www.eclipse.org/gmt/epsilon/doc/egl

The Eclipse Foundation, 2015d. Xtend, Java 10 Today! Available from http://www.eclipse.org/xtend

Tichy, M., Krause, C., Liebel, G., 2013. Detecting performance bad smells for Henshin model transformations. Workshop on the Analysis of Model Transformations, p.82-86.

Tran, V., Vanderdonckt, J., Kolp, M., *et al.*, 2009. Generating user interface from task, user and domain models. Int. Conf. on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services, p.19-26. [doi:10.1109/CENTRIC.2009.24]

van den Bergh, J., Luyten, K., Coninx, K., 2011. CAP3: context-sensitive abstract user interface specification. Proc. 3rd ACM SIGCHI Symp. on Engineering Interactive Computing Systems, p.31-40. [doi:10.1145/1996461.1996491]

Vanderdonckt, J., Limbourg, Q., Michotte, B., *et al.*, 2004. UsiXML: a user interface description language for specifying multimodal user interfaces. W3C Workshop on Multimodal Interaction, p.19-20.

W3C, 2010a. Model-Based UI XG Final Report. W3C Incubator Group Report. Available from http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504

W3C, 2010b. Model-Based User Interface (MBUI). W3C Incubator Group Report. Available from http://www.w3.org/2005/Incubator/model-based-ui

W3C, 2012. Media Queries. W3C Recommendation. Available from http://www.w3.org/TR/css3-mediaqueries

Zenger, M., Odersky, M., 2005. Independently extensible solutions to the expression problem. Int. Workshop on Foundations of Object-Oriented Languages, p.1-11.