Reusing UI elements with Model-Based User Interface Development

A. Delgado [a], A. Estepa [a], J.A. Troyano [b], R. Estepa [a]

[a] *Department of Telematics Engineering, University of Seville, Camino de los descubrimientos s/n., 41092 Seville, Spain*
[b] *Department of Computer Languages and Systems, University of Seville, Reina Mercedes s/n., 41012 Seville, Spain*

Keywords:
MBUID
Reuse
Software engineering
User Interface

A B S T R A C T

This paper introduces the potential for reusing UI elements in the context of Model-Based UI Development (MBUID) and provides guidance for future MBUID systems with enhanced reutilization capabilities. Our study is based upon the development of six inter-related projects with a specific MBUID environment which supports standard techniques for reuse such as parametrization and sub-specification, inclusion or shared repositories.

We analyze our experience and discuss the benefits and limitations of each technique supported by our MBUID environment. The system architecture, the structure and composition of UI elements and the models specification languages have a decisive impact on reusability. In our case, more than 40% of the elements defined in the UI specifications were reused, resulting in a reduction of 55% of the specification size. Inclusion, parametrization and sub-specification have facilitated modularity and internal reuse of UI specifications at development time, whereas the reuse of UI elements between applications has greatly benefited from sharing repositories of UI elements at run time.

1. Introduction

Model-Based User Interface Development describes the user interface (UI) through a collection of models which guide the UI development process. Such models can also drive the UI generation process in a (semi) automated fashion such as in model-driven engineering (MDE), which should reduce the work needed to develop UIs, resulting in a more productive software development process (Viana and Andrade, 2008; Meixner et al., 2010).

MBUID environments (MBUIDEs) have proliferated over the last decades, giving rise to a number of different toolsets and specification languages (Pinheiro da Silva, 2001; Guerrero-García et al., 2009; Meixner et al., 2011). However, in spite of the plethora of available approaches, no MBUIDE has experienced wide adop-tion by the software industry (Trætteberg, 2008; Molina, 2004). As envisaged by some authors (Pinheiro da Silva, 2001; Ahmed and Ashraf, 2007) the poor reusability of UI model specifications can be argued to partially explain this lack of interest from the industry. More recently, Meixner et al. (2011) have supported this idea, pointing also to the lack of harmonization between the MBUID and MDE worlds, as well as the absence of real-world usage and case studies as important challenges to be faced by MBUID systems in the future. This paper addresses some of these challenges, focusing specifically on reusability.

In general, software reuse increases productivity and software quality as reported in many industrial cases (Mohagheghi and Conradi, 2007), such benefits should also be present in the context of MBUID. Although some environments have been equipped with techniques to support reuse, issues and methods associated with the reuse of UI components and the benefit/cost associated have been shortly addressed in the MBUID community. One possible reason for this could be the complexity of the subject, which involves several inter-related questions such as (a) which UI fragments or models can be subjected to reuse; (b) what technical approaches can be used; and (c) how to assess the benefits of reusing. As stated in Meixner et al. (2011), the answer to these questions becomes even more complicated since emerging standards such as W3C task (Paternò et al., 2014) or Abstract UI models (Vanderdonckt et al., 2014) are not widely adopted yet, and there are scarce real-world usage and case studies that quantify the potential benefits of reusing UI assets.

This paper aims to be a first step in gaining insight into how elements from UI models can be reused. We provide a real-world usage case through the development of six applications with an environment that supports some standard reuse techniques such as parametrization and sub-specification, inclusion or shared repositories. We describe our experience and quantify the benefits of intra- and inter-project reuse of UI specifications. Based on our experience, we provide lessons learned that can be valid for other contexts and provide some advice on the development of future MBUIDEs with potentiated reuse features.

The remainder of this paper is as follows: Section 2 briefly introduces the main concepts of MBUID. Section 3 summarizes current reuse approaches already present in the context of MBUID. Sections 4 and 5 describe the main characteristics of the MBUIDE used in our case study and the techniques supported for reusing UI elements respectively. Section 6 introduces our study case. Section 7 addresses the results based upon our experience, discussing the use of each reuse technique and providing quantitative results of the benefits obtained. Section 8 provides an analysis of the main points of our results. Finally, Section 9 concludes the paper.

2. MBUID overview

MBUID offers an environment for developers to design and implement UIs in a professional, consistent and systematic way (Pinheiro da Silva, 2001; Meixner et al., 2011, 2010). MBUID is based on the idea that the UI can be fully modeled by a set of declarative models each addressing particular facets of the UI such as tasks and presentation. The specification of each model consists of an abstract description of the aspects pertaining to its domain by means of a so-called UI Description Language (Guerrero-García et al., 2009). Model specifications lead the UI development lifecycle and provide the basics for automatic UI generation.

Consensus on the set of models and languages for UI description has remained elusive in the past. Most MBUIDEs have defined their own languages and models (Guerrero-García et al., 2009; Meixner et al., 2011). However, some models were recurrently used by a number of environments in the early 00s (Pinheiro da Silva, 2001; Vanderdonckt et al., 2003):

- The *User model*, which specifies a hierarchical break-down of users in stereotypes that share a common role.
- The *Domain model*, used to define the objects accessible to users via the UI.
- The *Task model*, which describes the set of tasks that users are able to accomplish, its hierarchical decomposition and its temporal relations and conditions.
- The *Presentation model*, devoted to presentation aspects of the UI. It can, in turn, be decomposed into the *Abstract Presentation model*, dealing with abstract level descriptions of the structure and behavior of the UI objects, and the *Concrete Presentation model* that describes in detail the parts of the UI using modality-dependent (i.e. graphic and haptic) concrete interaction objects (Vanderdonckt and Bodart, 1993).
- The *Dialog model*, which defines the set of actions the user can carry out within various system states and the transition between these states. It links tasks with interaction elements forming a bridge between the Task and the Presentation models.

According to Meixner et al. (2011), the Task, Presentation and Dialog models can be considered the core models since they have direct influence on the content and appearance of the UI.

Different MBUID approaches can be related using the Cameleon Reference Framework (CRF) (Calvary et al., 2003). Since its definition in 2003, the CRF has become widely accepted in the HCI community as a reference for classifying UIs supporting multiple targets, or multiple contexts of use on the basis of a model-based approach (Meixner et al., 2011). The framework describes different layers of abstraction related to the model-based development of user interfaces:

- *Concepts-and-Tasks*: Specifies the hierarchies of tasks that need to be performed on/with domain objects (or domain concepts) for a particular interactive system (Meixner et al., 2010). Traditional Domain and Task models belong to this abstraction layer.
- *The Abstract UI*: Expresses the UI in terms of interaction units without making any reference to implementation in terms of interaction modalities or technological space (e.g. computing platform, programming or markup language) (Vanderdonckt et al., 2014). Abstract presentation or dialog models belong to this layer of abstraction.
- *The Concrete UI*: Describes concretely how the UI is perceived by the users using concrete interaction objects (Vanderdonckt and Bodart, 1993). These objects are modality-dependent but implementation-language-independent. Concrete Presentation models belong to this layer.
- *The Final UI*: Expresses the UI in terms of implementation-dependent source code. It can be represented in any UI programming or mark-up language (e.g. Java or HTML).

The CRF distinguishes between development and run time phases. In the development phase, initial model specifications are refined in successive steps. Ultimately, a Final UI expressed in source code is generated in a manual or automatic fashion from the concrete UI (Fonseca et al., 2010). Final UIs can then be interpreted or compiled as pre-computed UIs targeted for specific contexts of use (i.e. user, platform and/or environment) and plugged into an environment that supports dynamic adaptation to multiple targets at run time (Calvary et al., 2003).

3. Related work

Improving the reusability of model specifications in the context of MBUID has been addressed in the past. The main approaches found in the literature can be classified as:

- Reuse based on *the UI Description Language*: Some languages have foreseen the need to reuse fragments of specifications and have defined specific technical methods to deal with it. For instance, Hyatt et al. (2001) allows referencing specification fragments defined in the same document or included from an external document using the special processing instruction $<?\text{xul}-\text{overlay}?>$. In XICL (Sousa and Leite, 2005; de Sousa and Leite, 2006) the UI is made up of components which are somehow similar to classes. XICL allows the inclusion of components which can be extended through certain language tags and attributes (e.g. *extends*, *child*) in an analog way to object-oriented programming. In UIML (Abrams and Helms, 2004), frequently used specification fragments can be defined as templates using the $<\text{template}>$ tag, and reused in a flexible way (e.g. cascade, replace or union the referenced element). In addition, templates can receive parameters whose value will be passed when referencing. All previous approaches allow the developer to create a library of reusable assets, enabling the scope of the reuse to be internal (i.e. intra-project) or external (i.e. inter-projects).
- Reuse through *multiple transformations*: Third (TERESA Mori et al., 2003, XMobile Viana and Andrade, 2008) and fourth generation (e.g. MARIA Paternò et al., 2009, GUMMY Meskens et al., 2008) MBUIDEs are capable of generating multi-target UIs (Meixner et al., 2011). Therefore, a single UI model specification can be transformed multiple times targeted to different context of use which are defined for a set of users, hardware and software platforms, and physical environment (Calvary et al., 2003). This can be viewed as a kind of generative programming (Mohagheghi and Conradi, 2007). The scope of reuse is normally

Table 1
Approaches for the reuse of UI specifications.

Group	Approach	Main Technical Method	Scope (& granularity)
Language-based	XUL XICL UIML	Compositional Component-based, O.O. Compositional, templates	Internal or external (fine-grained)
Multi-Target	TERESA , XMobile, MARIA, GUMMY	Generative programming	Internal (coarse-grained)
Pattern-based	PD-MBUID, PIM PCB	Patterns Patterns, component-based	Internal or external (coarse-grained)

internal as reuse is restricted to new transformations of a specific UI lacking sufficient granularity to reuse just UI components.

- Reutilization through *pattern-based techniques*. Patterns can be viewed as abstract meta-models of frequent solutions which can be used to build UI specifications. Pattern-driven and model-based UI (PD-MBUI) (Ahmed and Ashraf, 2007) define a unifying framework where pattern-driven development is applied to building UI models. Tools such as PIM (patterns in modelling) (Radeke et al., 2007) can help the developer in this process. Patterns are also applied along with component-based reuse in Sinnig et al. (2005), where the authors propose to instantiate patterns related to models of the UI (e.g. task and presentation) and then use these instances as components of the UI. Patterns do not limit the scope of reuse, which can be either internal or external.

Table 1 provides a summary of the previous approaches, the main technical method for implementing reuse according to the generic classification in Mohagheghi and Conradi (2007), as well as the scope of the reuse and granularity of the assets.

Observe that although previous approaches have been implemented in real-world systems, little is known about experiences with the reuse process and obtained results. Thus, our work aims to be a first step in this field.

4. Description of our MBUIDE: WAINE

This section describes the system used in our case study: WAINE (Delgado et al., 2007), a basic MBUIDE that generates web applications based on the form paradigm.[1] WAINE was designed to facilitate systematic UI development to engineering students that were not acquainted with the plethora of current web-related technologies. A design goal was to simplify and speed up the development process by avoiding the need to define concrete aspects of the UI and pro-actively supporting the reuse of specifications.

4.1. *UI models in WAINE*

WAINE's *Domain model* describes the data that users handle via the UI and is specified through an Entity-Relationship Diagram (ERD). At a lower abstraction level, the following models (illustrated in Fig. 1) determine more directly the content and appearance of the UI.

- The *Presentation model*: Defines elements from the Abstract Presentation model. It has two major constructs: containers and forms. A form aggregates Abstract Interaction Objects (AIOs) such as fields (i.e. input/output interactors which displays and manipulate data), and action-launchers (i.e. controls) that allow the user to perform actions. Containers define the structure, content and basic behavior of an interaction unit (Vanderdonckt et al., 2014) (see Fig. 5). Containers can be of type *form* (to define an interaction unit composed of a single form), *split* (to create a spatial division in various areas), or *relation* (to create a spatial division in two areas that handle inter-related data). Containers use parameters to specify the content of each zone through a reference to a form or to another container (see Fig. 1). We will expand on forms, containers and parameters later in Section 4.3.
  A Concrete Presentation model is refined automatically from this model in a pre-defined way. However, each container, form or field can use its parameters or properties to override this default mapping and re-define some of its concrete aspects such as theme, style or concrete interactor. Thus, WAINE's Presentation model can also cover concrete-level definition to some extent.
- The *User model*: Users are categorized into groups according to their role. One group can have many users but each user only belongs to one group. Groups have specific views of the dialog model and security options.[2]
- The *Dialog model*: Includes a set of menu options accessible to users according to their group. Each option references an interaction unit to display identified by its top-level container or an action to be executed.
  There are events associated to the User and Presentation models (UM-Event and PM-Event respectively in Fig. 1) upon which occurrence an action can be triggered. For instance, onUserLogin and onUserLogout events are defined in the User model, form-related events such as onFormLoad and onFormUpdate are defined in the Presentation model.

4.2. *Architecture and final UI generation*

The UI is automatically generated following a run-time architecture (shown in Fig. 2). At the reception of a user action over the UI (1 – user request) the run-time engine generates the Final UI (2b), which is formed by HTML, CSS and Javascript and is sent (3) to the user browser. Three repositories are used by the run-time in this process (2a):

- *UI models*: Stores the instances of the UI models from Fig. 1.
- *UI data*: Handles the data referenced by the UI.
- *Customization*: Composed of a collection of files that deal with concrete aspects of the UI (e.g. fonts, borders, and colors),

[1] WAINE is not multi-modal since it is targeted only for the graphical modality.

[2] A mechanism based on Access Control Lists is implemented to restrict the access of a user or group to a specific form or container. However, this is not shown in Fig. 1 for the sake of clarity since it does not contribute to the goal of the paper.

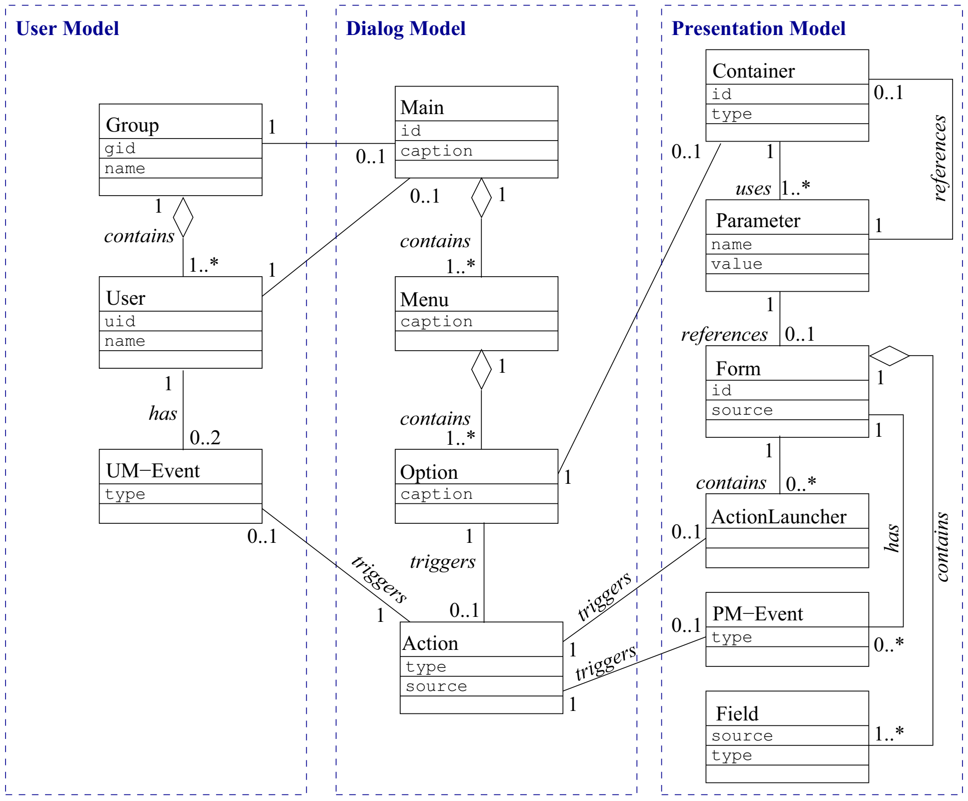


Fig. 1. WAINE UI models.

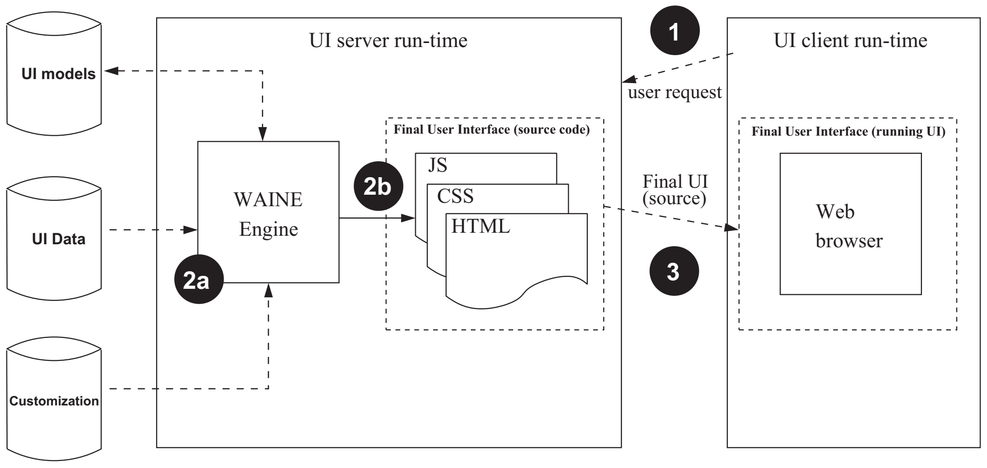


Fig. 2. Run-time implementation architecture.

database access configuration[3] and other implementation-level aspects related to widgets, authentication methods and actions.

When an application is initiated, the run-time engine automatically generates a login form for authentication. Once the user is authenticated, the engine fetches the objects from the Dialog model that correspond to the user (or his group) and displays the menu options. At the user selection of a menu option the run-time engine will either execute an action or generate the code of an interaction unit. In the latter case, the UI models and UI data repositories are retrieved to get the AIOs involved (e.g. containers and forms) and the linked data. Then, the concrete presentation is composed using the default abstract-to-concrete mapping specified in the customization repository.[4] Ultimately, the run-time generates the Final UI. This process is repeated until the user logs out.

4.3. UI development life cycle

The activities of the development process finish with the provision of the three repositories used by the run-time engine. Fig. 3 illustrates the main steps involved in this process as well as their relationship with the levels of abstraction

[3] A local configuration file indicates to the run-time the connection method for each database used, enabling transparent location of UI objects by sequential search of the object among the databases listed.

[4] As mentioner earlier, concrete presentation aspects defined by AIOs properties will prevail over this default mapping.

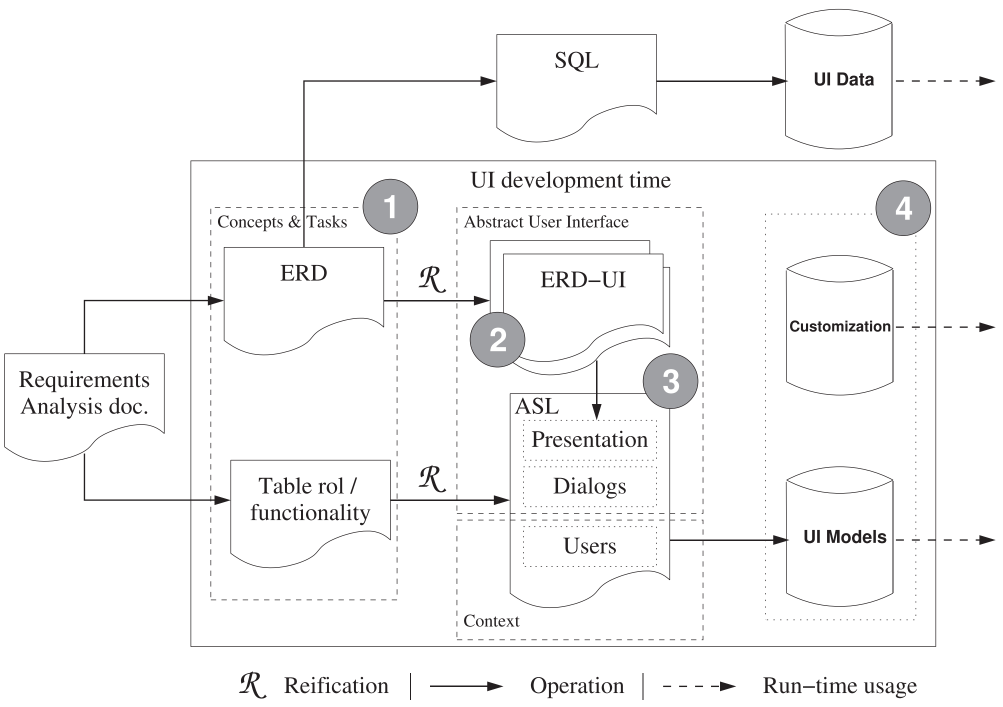


Fig. 3. Main steps in WAINE development life cycle.

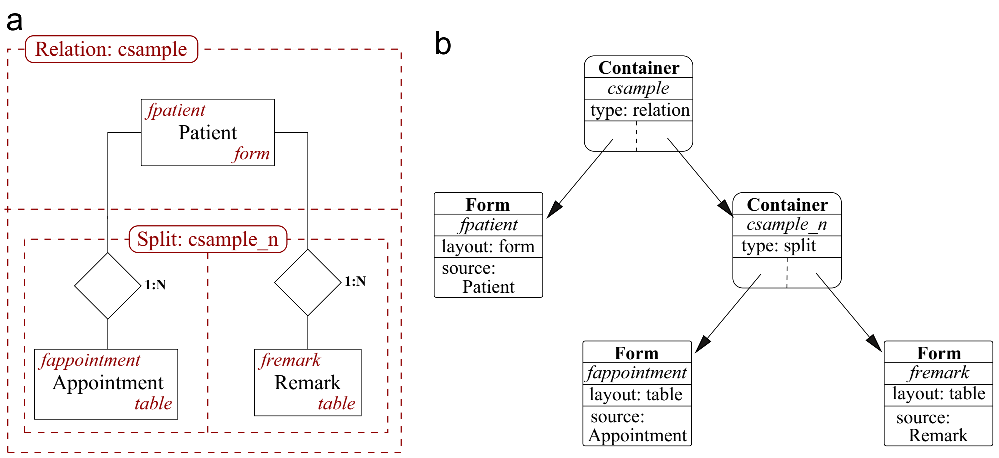
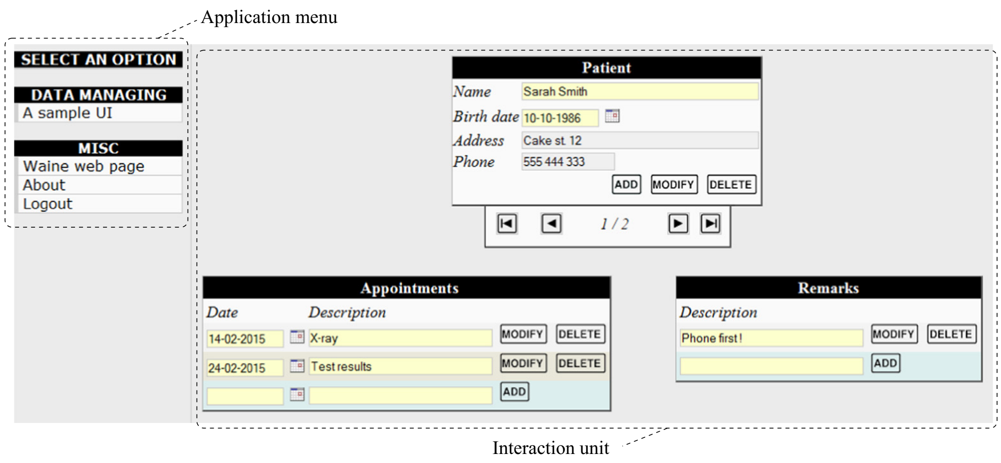


Fig. 4. Example of annotation of ERD diagram.

from the CRF (in dashed line). The process begins with a requirement analysis document that includes functionalities, user roles and a sketch of UIs and reports. The following steps are as follows:

1. *Concepts modeling*: From the requirement analysis two documents are generated: (a) an ERD that captures the concepts handled by the application (WAINE domain model) and (b) a table that specifies needed actions and interaction units for each possible role (i.e. role functionalities).
2. *ERD-UI annotation*: This reificatory process specifies each interaction unit in the application by annotating the ERD from step 1. To do so, it enriches the portion of the ERD affected by the interaction unit annotating its structure and layout. To illustrate and expand on this process we develop a simple example application to manage patient appointments through a single interaction unit.
   The visual structure of the interaction unit in our example is composed of two zones horizontally divided that hold a data relationship one-to-many. Fig. 4(a) shows how this is annotated in the ERD with a dashed line around the implied entities and the container type (Relation) and id (*csample*). The top area will allocate a form that handles data of the entity Patient. This is annotated by writing the form identifier (i.e. *fpatient*) and layout (e.g. form, table, grid, and list) in the corners of the own Entity. In the bottom zone of *csample* we divide the space vertically by using a new container: *csample_n* of type Split. Forms to deal with appointments (*fappointment*) and remarks (*fremark*) related to each patient will be defined on the left and right sides respectively.
   The behavior of the interaction unit is defined by the container *csample*. If the data element in the *one* zone changes (occurrence of Patient in our example) the data elements in the *many* zone (occurrences of entities Appointment and Remark) are automatically recalculated. This synchronization process is facilitated due to the hierarchical relationship between the UI constructs illustrated in Fig. 4 (b).



Sarah Smith
10-10-1986
Cake st 12
555 444 333
14-02-2015 X-ray
24-02-2015 Test results
Phone first !

Fig. 5. Example UI.

The output of this step is a collection of annotated ERD-UI documents (one for each interaction unit).

3. *ASL specification*: In this activity, the developer takes the table role/functionality and ERD-UI diagrams, and creates a document written in an XML-based language termed ASL (Delgado et al., 2007) where instances of the classes from Fig. 1 (UI models) are specified. Appendix A provides a specification of the interaction unit previously defined (see Fig. 4) including the definition of a group with one user and the elements from the Dialog model generated from the table role/functionality (tags *group*, *user*, *main*, *menu* and *option*). Observe from the ASL specification in Appendix A that:
   - The majority of the specification lines are devoted to the definition of the Presentation model.
   - A *form* definition requires some attributes to be set such as identifier, source and caption, and includes the definition of the primary key (tag *key*) and optionally foreign key (tag *fkey*), as well as its composing fields which in turn encompasses the definition of its source, abstract interactor (e.g. string, date), etc.
   - Each container uses a variable number of parameters. Parameters can be classified into three categories:
     - *Structural*: Used to define a spatial structure (e.g. the *form_split* tag is used in the container *csample* to create a division in two rows assigning 20% of the available space to the top one).
     - *Content*: Describing the content for each zone (e.g. tags *containerid*, *formid* reference a content for a zone indicated by the attribute *ord*) and its basic properties (e.g. *form_layout*).
     - *Customization*: Oriented to overwrite values defined in a form that is referenced. Section 5 includes an example of these kind of parameters.
4. *Generation of repositories and customization*: The development process finishes with the creation of the three repositories used by the run-time engine. The UI models repository is automatically generated by a tool which checks the syntax of the ASL document and transforms it into a populated database. The UI data repository is automatically generated from the ERD. Finally, a default customization repository is also automatically generated for the new application. This repository includes information to refine the concrete presentation of the UI such as Listings 1 (mapping abstract to concrete interactor) and 2 (colors and styles), as well as other configuration information such as databases and paths to use. Developers can overwrite or add new objects to this repository to customize the application. Fig. 5 shows the default aspect generated for the ASL document in Appendix A.

Listing 1. Abstract-to-concrete interactors mapping.

```
integer.widget='editbox';
float.widget ='editbox';
string.widget ='editbox';
time.widget ='timebox';
date.widget ='datebox';
text.widget ='textbox';
image.widget ='imagebox';
```

Listing 2. Colors and styles configuration.

```
$BODYATTR=['bgcolor','background','text'];
$BODYVAL =['#00659c','img/bg.png','#005989'];
$MENUATTR=['class','bgcolor'];
$MENUVAL =['clmenu','#e7efff'];
$OPTATTR=['class','bgcolor'];
$OPTVAL =['clopt','#efeff7'];
$FRMTITATTR=['bgcolor','style'];
$FRMTITVAL =['#c6dbff','border: 1px solid black;'];
```

5. Reuse techniques supported by WAINE

One of the design goals for WAINE was to increase development productivity, which lead us to proactively support reutilization techniques. We have implemented compositional techniques based on the UI description language (according to our classification in Section 3) to reuse fragments of ASL specifications. In addition, we also have implemented the capability to reuse the repositories among various applications which is a novel way to support the reuse of transformed specifications.

5.1. Reuse based on the language

The techniques supported by ASL are as follows:

- *Inclusion* (*I*): ASL uses the W3C standard XInclude (Marsh et al., 2006) instead of a language-specific method (e.g. XUL or UIML), which requires the implementation of the *include* and *fallback*[5] tags (see Fig. 6). XInclude can be used to insert fragments of specifications stored in an external file into the ASL document. Additionally, it can also be used to modularize large specifications.

[5] The fallback mechanism contains alternate content to be used if the requested resource cannot be found.

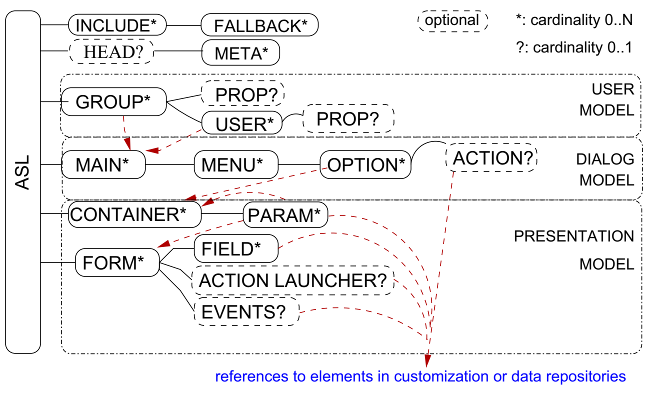


Fig. 6. References and parameters in ASL structure.

- *Sub-specification and Parametrization* (*S,P*): ASL allows one to reference UI elements and adapt such elements through parametrization. To gain insight into how (S,P) are implemented Fig. 6 shows the structured hierarchy of the main ASL tags and its inter-relation. Dashed arrows indicate sub-specification. For example, a container can be referenced from menu options, a form can be referenced from the parameters of different containers, and so on. Observe from the syntax and structure of ASL that only top-level tags (i.e. main, container or forms) and objects in the customization and data repositories are reusable through sub-specification.
  As mentioned earlier, containers can use parameters to override any property of their composing forms. This supports the adaptation of the reused asset. For instance, Fig. 7 shows how the form *fpatient* previously defined in our example (see Appendix A and Fig. 5) can be reused changing its layout from default (*form*) to *table*, removing its fourth field, forbidding edition in the first and third fields, and suppressing the action launcher for the insertion of new tuples (add button in Fig. 5).

Note that (I) can be combined with (S,P) to use a library with useful fragments of specifications to be included in various projects.

5.2. Reuse based on run-time repositories

In our MBUIDE the run-time does not directly access the UI specifications but repositories that contain transformed versions of such specifications. If these repositories were accessed by various run-time engines various applications could share any elements from any of their UI repositories.

We have implemented in the run-time the capability to perform a sequential search of a referenced element among a list of repositories which can or cannot belong to the application. This can be viewed as a comfortable way of inclusion where the developer does not formally include or know the source of the asset but just references it by its name. The main constraints are (a) the element identifier must not be duplicated in different repositories (the first hit will be selected in such case) and (b) access to shared repositories must be fast enough to avoid impairing the UI response time.

Sharing repositories of UI objects between run-times (i.e. applications) can employ one of the following strategies illustrated in Fig. 8:

- *Central repository* (*C*): A central shared repository can store UI objects common to various applications. In addition, each application could have its local repositories storing its specific elements as illustrated in Fig. 8(a).
- *Federated repository* (*F*): Federation allows objects residing in different locations to be linked and accessed in a transparent way for users, as if those objects where residing in the same location. A new UI can be composed by federating UI fragments from other applications. The resultant federated UI will obtain assets from various repositories which belong to independent and autonomous applications as illustrated in Fig. 8(b).

Fig. 9 illustrates the methods supported and their context of use within the development process.

6. Case study

This section aims to provide a real-world experience where reutilization is studied after developing six projects with WAINE. We believe that our experience provides interesting insights into the technical approaches applied, as well as the factors that can impact the reusability of the assets.

Our University demanded a common development environment for a set of inter-related projects in a large School of Engineering where 500 lecturers and 5400 students from 12 different degrees[6] share common resources such as 112 rooms, 856 subjects, 16 screen panels and a number of activities like exams, seminaries, lectures, and scholarships. A total of six different applications were defined to manage different aspects of the aforementioned resources. Table 2 shows a list of the projects sorted by development date. Applications were developed sequentially. For each application columns 2–5 provide data related to the first step of our development process: number of roles (i.e. user groups such as professor and student), total functionalities, number of Interaction Units (IUs) and number of database tables and views. Columns 6 and 7 show the duration of the project in weeks (a week is equivalent to 40 h of work), and the size of the ASL specification in lines of code (LOC) (step 4 in the development process). We consider A1 as a large project and A2–A6 as small projects. The last column describes the purpose of each application. Fig. 10 shows a screenshot of the application A6.

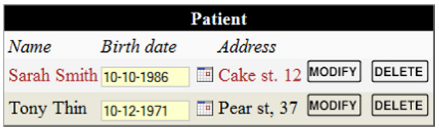
The applications listed in Table 2 exhibit common characteristics such as roles, look & feel, common forms and data. Our initial motivation was not to perform an exhaustive experiment on reuse, but to exploit the potential commonality to reduce the development effort by reusing ad hoc UI components previously developed. Note that we did not start with project A_i requirements until project A_{i-1} was finished, which implies that we did not know a priori what assets would be needed in future projects.

On each project, new UI assets were generally developed as needed without a specific design for reuse. Consequently, the repertoire of assets available for reuse was mostly built along with the first application (note that A1 is substantially greater than A2–A6) and grew slightly with each application developed.

The first application (A1) was developed by a high experienced team composed of the first two authors (average experience over five years). The following applications (A2–A6) were developed by teams composed of a single graduate-level student (low experience) closely tutored by the main author. Each student developed one application. All students exhibited comparable skills and received a three day training about WAINE to become acquainted with the development process and ASL coding through the development of a sample application. All techniques for reuse supported by WAINE were practiced in the training sessions. The tutor lead the development steps up to 2 (i.e. requirements, ERD, role/functionalities). Students participated passively in these steps, but were responsible for writing ASL specifications (step

[6] Data from the academic year 2013.

```
<container id="cparam_sample" type="form">
  <param name="formid" value="fpatient"/>
  <param name="form_layout" value="table"/>
  <param name="fields_remove" value="4"/>
  <param name="fields_readonly" value="1:3"/>
  <param name="button_insert" value="NO"/>
</container>
```



Patient				
Name	*Birth date*	*Address*		
Sarah Smith	10-10-1986	Cake st. 12	MODIFY	DELETE
Tony Thin	10-12-1971	Pear st, 37	MODIFY	DELETE

Fig. 7. Reusing the form *fpatient* with (S,P).

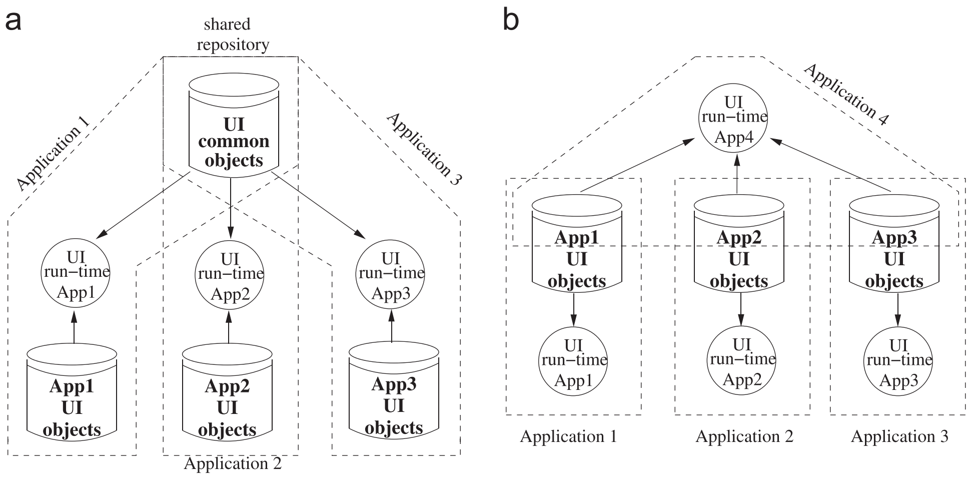


Fig. 8. Centralized (a) and Federated (b) architectures for sharing UI objects between various UI run-times.

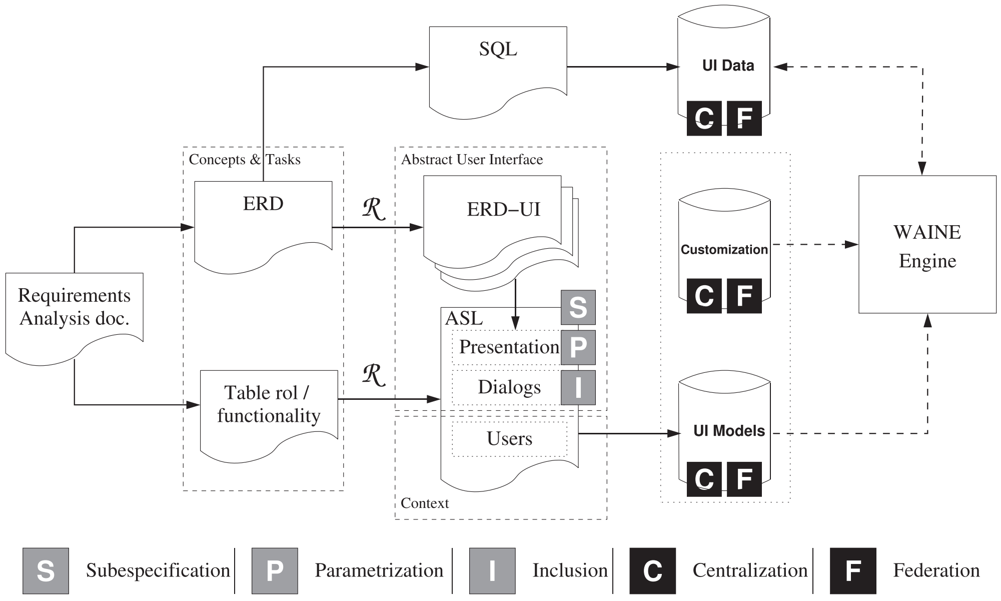
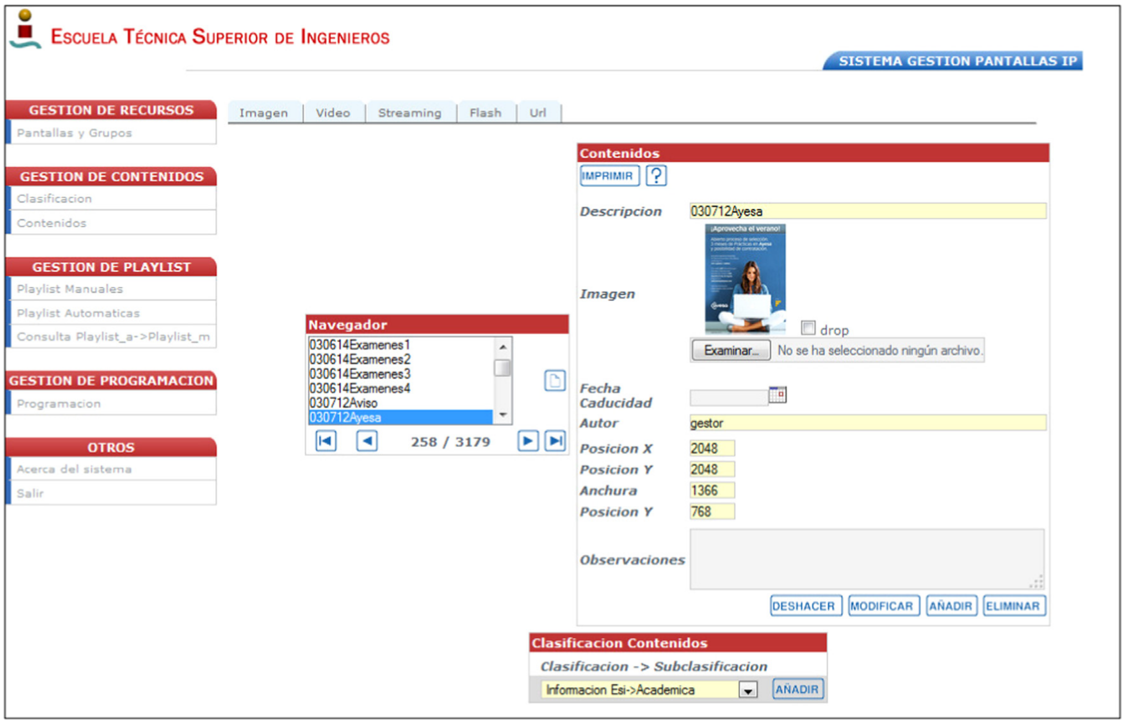


Fig. 9. Context of use of the techniques supported.

Table 2
List of applications developed.

App.	Roles	Functionalities	IUs	Tables Views	Duration (weeks)	ASL size (LOC)	Purpose
(A1) reservas	11	204	188	98	15	5696	Room reservation management
(A2) inveq	2	24	23	40	4	578	Inventory management
(A3) board	5	26	22	10	5	768	Departmental advertisement
(A4) pracemp	1	20	18	20	4	521	Scholarships management
(A5) relext	1	11	10	16	4	239	Foreign affairs management
(A6) scrplan	2	13	11	24	6	904	Screen panels advertisement

3) for the assigned project. Assets developed in previous projects were known to the tutor, who did ad hoc search, selection and evaluation of the assets to be reused. Students implemented reuse as indicated by the tutor and occasionally implemented opportunistic internal reuse over assets that they had previously developed. Each student was supervised by the tutor twice a week.



Navegador:
030614Examenes1
030614Examenes2
030614Examenes3
030614Examenes4
030712Aviso
030712Ayesa
258 / 3179

Descripcion	030712Ayesa
Imagen	drop; No se ha seleccionado ningún archivo.
Fecha Caducidad	
Autor	gestor
Posicion X	2048
Posicion Y	2048
Anchura	1366
Posicion Y	768
Observaciones	

Clasificacion -> Subclasificacion: Informacion Esi->Academica

Fig. 10. Screenshot of *scrplan* (A6).

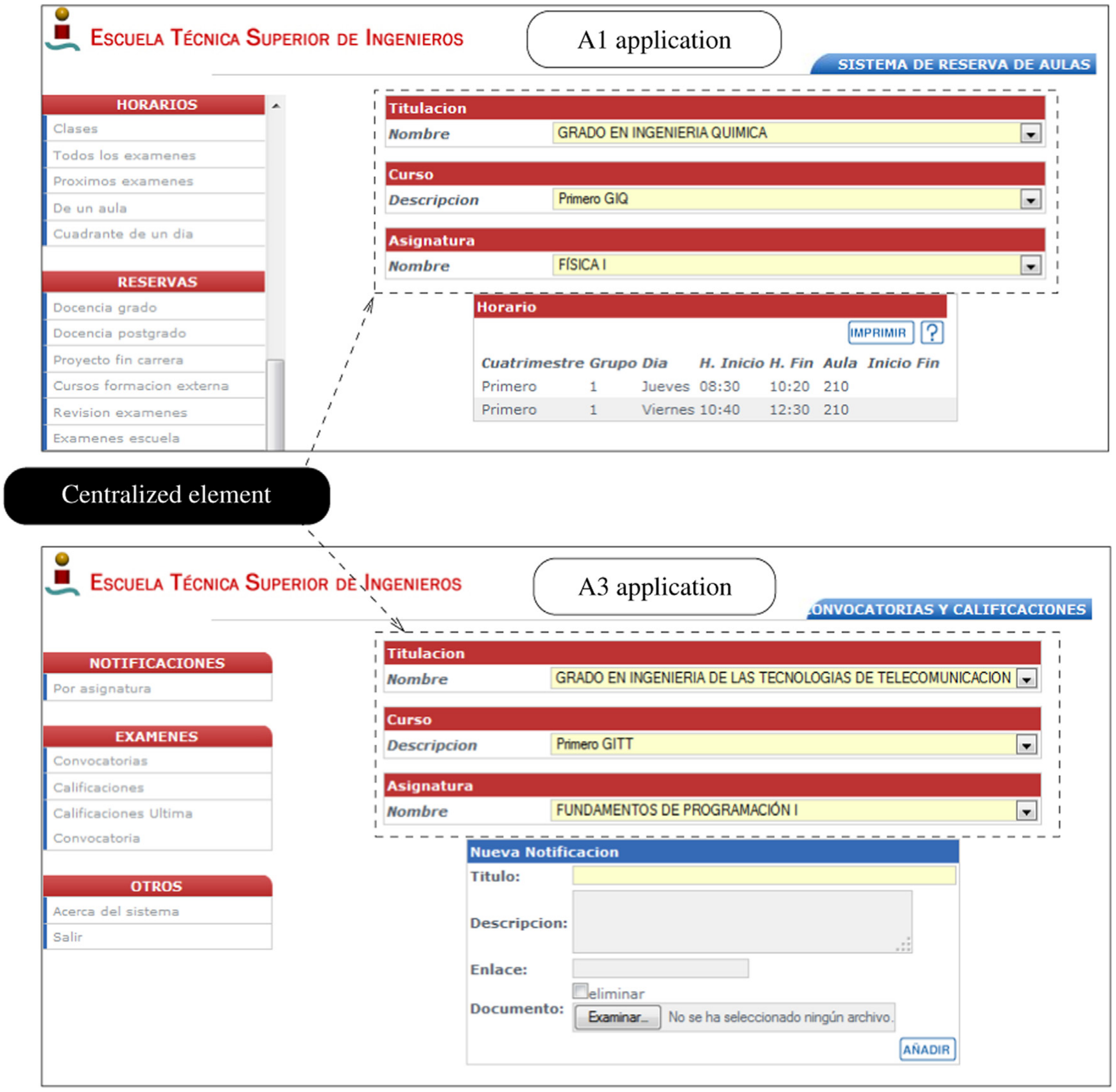
7. Results based upon our experience reusing UI assets

Software reuse is a complex subject that encompasses all phases of the software development life cycle (Basili et al., 1995; Gill, 2003; Kontio, 1996) and requires the careful use of metrics (Mascena et al., 2005; Baldassarre et al., 2005). An excellent review of industrial cases can be found in Mohagheghi and Conradi (2007). This section does not intend to offer an exhaustive analysis of software reuse but to serve as an illustrative example of our particular experience reusing UI assets with the techniques implemented by WAINE and quantify some of its benefits after the development of the projects listed in Table 2.

7.1. About the application of each technique

- *XInclude* (*I*): We have mostly used this standard inclusion mechanism to modularize large ASL documents for better maintenance and, to a lesser extent, to include UI elements common to various applications. In particular during the development of A1, only two interaction units a priori identified as common to all applications (referenced from the menu options *user management* and *about*) were defined in their own files and included in all projects. This scarce use of inclusion can be traced back to the fact that we did not count on a specific library of reusable objects (except for the aforementioned) but the ad hoc reuse of existing assets from previous projects.
- *Sub-specification and Parametrization* (*S,P*): The use of sub-specification has allowed elements from the same document or from other applications (when used along with run-time repository sharing) to be reused verbatim (without modification). However, we found ourselves limited in reusing elements from the Dialog model since its asset were non-adaptable, coarse-grained elements that included the full set of menu options associated to a specific role, and it is unlikely that different roles have the same set of functionalities. Conversely, assets with a less hierarchical structure have been frequently reused within the Presentation model (e.g. a form or container definition referenced by other containers).
  Parametrization has made reuse more attractive since original assets can be adapted. This has notably increased the reusability of forms. For example, in A1 a single form definition (form id *form_subjects* with ten fields to handle personal information) was internally referenced up to twelve times from different containers using parameters to obtain different variants.
- *Sharing run-time repositories* (*C,F*): We have always[7] applied this technique in combination with (S,P) for inter-project reuse. This is attributable to various reasons: (a) all applications shared a common infrastructure (school data center) which eases management and provides good response time; (b) it saved us the need to use a formal inclusion mechanism during ASL coding; and (c) as mentioned earlier, we had not created specific libraries of assets except for the two aforementioned.
  The run-times were configured to access not only their own three repositories, but also the repositories of those applications where referenced elements had been defined (mostly A1 repositories). This gave us the possibility of reusing fully functional components of the UI between applications in a flexible way (i.e. abstract and/or concrete presentation and/or data interfaced). It was particularly useful in frequently used forms defined in A1 and used by the rest such as subject selection, room location, etc. In these cases, A1 repositories acted as central repositories for the other applications. Fig. 11 shows an example where a container for subject selection (composed of three inter-related selectors for degree, academic year and subject) defined by A1 is also used by A3. Centralization has also improved maintenance since changes in adaptable assets can be immediately noticed by all connected applications or ignored through parameters in those applications that do not need them.
  Less frequently than centralization, we have also reused UI

[7] The only exceptions to this were the containers and forms related to *user management* and *about* dialogs.



Titulacion

Nombre	GRADO EN INGENIERIA QUIMICA

Curso

Descripcion	Primero GIQ

Asignatura

Nombre	FÍSICA I

Horario

Cuatrimestre	*Grupo*	*Dia*	*H. Inicio*	*H. Fin*	*Aula*	*Inicio*	*Fin*
Primero	1	Jueves	08:30	10:20	210		
Primero	1	Viernes	10:40	12:30	210		

Titulacion

Nombre	GRADO EN INGENIERIA DE LAS TECNOLOGIAS DE TELECOMUNICACION

Curso

Descripcion	Primero GITT

Asignatura

Nombre	FUNDAMENTOS DE PROGRAMACIÓN I

Nueva Notificacion

Titulo:	
Descripcion:	
Enlace:	
	eliminar
Documento:	No se ha seleccionado ningún archivo.

Fig. 11. Example of centralized form shared by two applications.

Table 3
Use of techniques.

Technique	Reuse scope	Function	Frequency
(I)	–	Modularize specification	–
(S,P)	Internal	Reuse verbatim and adapt parametrized assets	Always
(I)+(S,P)	External	Reuse assets defined in external ASL document	Seldom used
(S,P)+(C,F)	External	Reuse assets from repositories of any application	Always

elements by federating repositories when some of the interaction units of an application were maintained by other applications. For instance, A4 interaction units show information about destinations (e.g. countries and regions) directly managed by A5 as well as information about professors, departments or degrees directly managed by A1. Therefore, A4 was not accountable for changes in the data or presentation of such information which depended on their respective management domains.

Table 3 summarizes the final application of each available technique.

7.2. *Quantitative results of the overall reuse*

It is worthwhile recalling that WAINE reusable components (UI assets) can be:

- ASL top-level tags from the User (< group >), Dialog (< main > menu) or Presentation models (< container > or < form > , see Fig. 6).
- Database tables or views interfaced by the UI.
- Individual files from the customization repository related to concrete presentation such as style definitions and configuration files.

The metric *amount of reuse* (Frakes and Terry, 1996) counts the portion of assets defined in a project used more than once (either within the same project or reused in other projects). Table 4 shows the number of assets defined (row D), the number of reused assets (row R) and the amount of reuse (column %) for each application and WAINE repository. The last row and column provide total numbers for each application and row respectively.

Table 4 shows a comprehensive description of our experience in which we highlight the following ideas:

- The application A1 defines 65% of the total assets developed; 61% of its UI assets have been reused either internally or by other applications. Almost all concrete presentation assets in its customization repository have been reused by the rest (A2–A6), getting a homogeneous look in all corporate applications.
- Elements from the Dialog and User models have been seldom reused. This can be traced back to the own structure of those assets in their respective models: single coarse-grained and not adaptable. Additionally, in the case of the User model, groups cannot be referenced from other models, leaving inclusion as unique reuse method. At any rate, dialogs and users account a minimal portion of the overall assets defined in ASL (7%).
- The Presentation model accounts for a major part of the assets defined in ASL (93%). In the case of A1, almost 57% of its Presentation model assets have been reused. Data interfaced by the UI have also been reused, but always along with the reuse of the linked abstract interaction objects.

The previous reasons justify why we focus our interest on the Presentation model for the remainder of this section. We believe that a deeper study of the reuse of elements in the customization or UI data repositories is not interesting in our particular case due to (a) A1 customization repository acts as a central repository almost fully reused by the rest, and therefore, the benefit is easily predictable; and (b) data reuse has already been thoroughly studied in literature (Frakes and Kang, 2005), and in our context it always takes place through the Presentation model.

7.3. *Reuse of the presentation model*

The rest of this section provides a quantitative analysis of the scope of the reuse, the overhead introduced when assets are adapted, and how the reuse of this central model in WAINE has improved software development productivity.

7.3.1. *Reuse scope*

To observe whether reuse is predominantly internal or external in each project, we use the *Reuse Level* (*RL*) metric (Frakes and Terry, 1996), defined as the ratio between the number of objects reused in a system versus the number of different objects that a system utilizes. Table 5 presents for each application: the number of different Presentation model assets used (L), the number of elements internally reused (i.e. defined and reused in the same project) (M), the number of elements not defined by the application but leased from other applications (E), the internal reuse level (internal RL), the external reuse level (external RL) and the total reuse level (total RL).

As expected, Table 5 shows that intra-project reuse is clearly dominant in the first development A1; the other projects do not show a general tendency. For instance, A4 or A5 benefit from inter-project reuse (mostly from A1) while internal reuse is dominant in those applications that are less integrated with the rest. This is consistent with the findings described in Mohagheghi and Conradi (2007) for general software reuse in industrial projects: “in small-scale studies, reused assets are internal or external, while in the medium to large-scale studies, reused assets are all internal”. Our results also suggest that a high total reuse level can be achieved by external reuse when UIs of small applications (e.g. A4, A5) have high commonality with a large application, or by internal reuse in a large application such as A1.

7.3.2. *Adaptation of the assets*

Whenever a form was reused adaptation was carried out. Identifying the right form and adaptation takes some effort that unfortunately we have not measured, but it is worthy to compare the forms size in terms of ASL lines of code (LOC) versus the size of the *adaptation overhead* (i.e. counting one line of code for each parameter used to adapt the original asset) each time that a form has been reused. Fig. 12 shows the average size of the reused forms (categorized by size) versus the average overhead used to adapt them. We can conclude that the adaptation overhead is always less than the size of the adapted item, and that it grows at a lower rate than the asset size. In other words, the bigger the asset, the more *productive* its reuse is in terms of saving lines of code.

Table 6 shows the adaptation overhead by project. Columns two to four show the aggregated size of the forms, the overall adaptation overhead in LOCs and in percentage respectively. Relating these results to those from Table 5 we can observe that projects where reuse is predominantly internal (A1, A2, A6) have a lower adaptation overhead than projects where reuse is predominantly external (A4, A5). This supports the idea that adapting own assets has a lower overhead than adapting assets defined by other projects.

7.3.3. *Benefits of the reuse*

Cost-benefit analysis states that benefit derived from reuse can be expressed by the equation (Gill, 2003):

$$R_{save} = [C_{no\_reuse} - C_{reuse}]/C_{no\_reuse}. \quad (1)$$

where C_{no_reuse} is the cost of developing an application without reusing assets and C_{reuse} is the cost of developing an application reusing assets (which includes the cost of reusing).

Taking ASL code lines (LOC) as a rough metric to estimate the development cost in WAINE, we can do reverse engineering and estimate C_{no_reuse} by recalculating the number of code lines that would have been written if reuse had not taken place. This can be readily done by processing the ASL specification documents and

Table 4
Assets defined and reused per application and UI model.

Repository		A1	A2	A3	A4	A5	A6	TOTAL
UI data	**D**	98	40	10	20	16	24	208
	R	17	2	5	2	3	1	30
	%	17.3	5.0	50.0	10.0	18.8	4.2	14.4
UI models (ASL)								
(User model)	**D**	11	2	5	1	1	2	22
	R	0	0	0	0	0	0	0
	%	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(Dialog model)	**D**	11	1	4	1	1	2	20
	R	0	0	1	0	0	0	1
	%	0.0	0.0	25.0	0.0	0.0	0.0	5.0
(Presentation model)	**D**	346	52	50	25	19	54	546
	R	197	12	9	6	4	14	242
	%	56.9	23.1	18.0	24.0	21.1	25.9	44.3
Customization	**D**	190	3	4	3	3	4	207
	R	188	0	1	0	0	1	190
	%	98.9	0.0	25.0	0.0	0.0	25.0	91.8
TOTAL	**D**	656	98	73	50	40	86	1003
	R	402	14	16	8	7	16	463
	%	**61.3**	**14.3**	**21.9**	**16.0**	**17.5**	**18.6**	**46.2**

Table 5
Reuse level of the presentation model of each application.

Project	*L*	*M*	*E*	Internal RL	External RL	Total RL
A1	350	197	4	56.3	1.1	57.4
A2	54	12	2	22.2	3.7	25.9
A3	64	9	14	14.1	21.9	35.9
A4	44	6	19	13.6	43.2	56.8
A5	29	4	10	13.8	34.5	48.3
A6	58	14	4	24.1	6.9	31.0

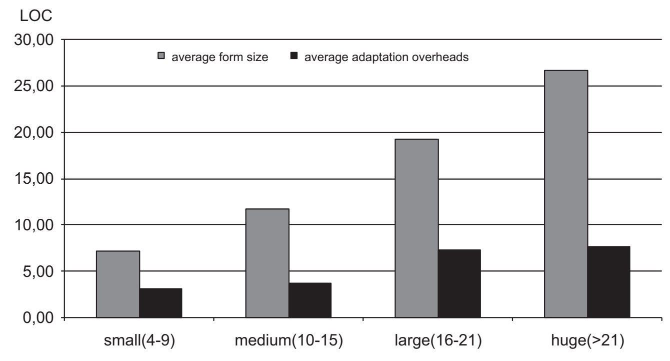


Fig. 12. Parametrization overhead by asset size.

Table 6
Overhead of adapting assets on each application.

Project	Total size of forms (LOC)	Parametrization overhead (LOC)	Parametrization overhead (%)
A1	2.021	359	17.8
A2	230	30	13.0
A3	212	70	33.0
A4	228	60	26.3
A5	91	24	26.4
A6	437	73	16.7
Average	536	103	22.2

updating a code lines counter every time a container or form is reused by simply adding its size and subtracting the LOCs written for its reuse (including the parametrization overhead), which we assume to be the cost of the reuse method. Table 7 shows for each application the LOC actually written (C_{reuse}), the LOC that would had been necessary to write without reusing forms and containers C_{no_reuse}, and the cost benefit of such reuse R_{save} as defined in Eq. (1). The last two columns show the duration of the ASL coding activity and an estimated duration of such activity without reuse respectively according to each project's productivity (i.e. the ASL size divided by the duration of the ASL encoding phase).

From Table 7 we can observe that reusing UI assets has reduced the size of the Presentation model specifications an average of 55%, peaking at 76% for our largest application A1. If we add the specifications of all applications (total row) we have attained a total benefit of 71% in terms of LOC savings and an estimated reduction of 66% in the duration of ASL coding.

7.3.4. *Other considerations*

The benefits of software reuse include improvements in productivity, quality and maintenance (Mohagheghi and Conradi, 2007). Unfortunately, we have not been able to collect data about quality. The only facts known after two years of operation and maintenance are five modifications of existing forms or containers have been made; 80% of these modifications affected assets in A1 which were being internally reused; twelve new assets were also created from which one-third were done by reusing existing assets. This suggests that the share of reuse obtained during development could also have some impact during maintenance.

8. Lessons learned and open issues

Our results are influenced by our context (i.e. projects, reuse process and MBUIDE). However, we believe that some general ideas can be used by the MBUID community in order to design new systems or models with enhanced reusability.

Table 7
Benefits of reuse per application in the presentation model.

Project	Presentation model size (LOC)	Estimated size without reuse (LOC)	R_{save}	ASL coding duration (w)	Estimated duration without reuse (w)
A1	5.335	22.225	0.76	10.5	42.1
A2	532	812	0.34	2.5	3.8
A3	689	1.636	0.58	2.8	6.6
A4	483	1250	0.61	2.1	5.4
A5	219	525	0.58	2.7	6.5
A6	865	1.463	0.41	4.3	7.3
Total	8.123	27.911	0.71	24.4	71.8
Average	1.354	4.652	0.55	4.1	11.9

8.1. *On the reusability of UI models through sub-specification*

UI models define elements (associated to top-level tags in XML-based languages) potentially reusable via sub-specification. These elements can be referenced from other elements that belong to the same model (intra-model reference) or from other related models (inter-model reference). According to our experience, intra-model sub-specification improves significantly the reusability of a model, especially if assets can also be adapted. To expand on this, we can analyze our Dialog model and figure out the consequences of changing the structure of its reusable asset.

Fig. 13(a) illustrates the present situation of the Dialog model in WAINE. Single monolithic elements (i.e. a single top-level tag < main > which includes a nested definition of its components – see Appendix A) can only be referenced from the User model. A difference in just one attribute of the < main > element such as its caption forces the developer to specify two complete assets. However, if assets within that model were composed of referable parts as in Fig. 13(b), intra-model reuse could take place eventually letting higher-level components reuse lower-level components through intra-model sub-specification. This increases the number of reusable assets in the model. Naturally, it requires changes in the language syntax. For example, in order to make (b) possible, the structure and syntax of ASL should define < main > and < menu > as top-level elements and let the latter be referenced from the former in an analog way to containers and forms. Intra-model reuse, along with adaptation, has been key in the high reusability of our Presentation model.

Of course, it can be argued that high fragmentation (for example three levels of referenced parts in the previous example main $\rightarrow$ menu $\rightarrow$ option) also complicates the process of reuse and adds more overhead. The cost of reusing is also influenced by factors such as time spent searching, screening and evaluating the items to be reused (Kontio, 1996) not assessed in our particular case. However, in industrial environments with large projects these factors should be taken into account. Therefore, a balance needs to be found between model fragmentation and the complication and cost of the reuse process.

8.2. *On the size and adaptability of the assets*

As a general principle, the bigger an asset is, the more profitable its reuse is. However, big coarse-grained assets can be less reusable if they are not adaptable. Adaptability notably increases the reusability of any asset according to our experience. But it requires assets to be parametrized and their structure to be known. On the contrary, the smaller an asset is, the less profitable its reuse. For instance, although 60% of the forms reused in our case had a small size (4–9) they only produced a share of 30% of

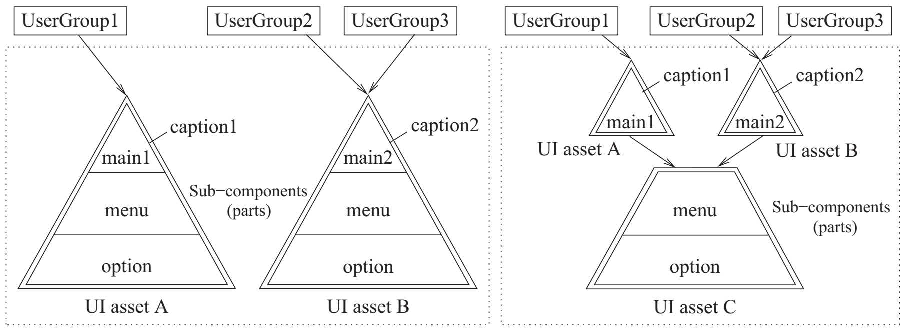


Fig. 13. UI Model fragmentation.

the benefit in terms of saving lines of code attributable to reusing forms.

In general a less fragmented model tends to use bigger assets and viceversa. Consequently, we can conclude that there is a trade-off between the level of composition of a UI Model, the size of its assets, and the benefit of reusing them.

8.3. *On the techniques utilized*

Different techniques apply to assets of different nature: XInclude is able to reuse XML fragments of any kind (even a single XML line); sub-specification requires the reuse of a complete well-formed UI asset; repositories allow elements of a nature different from XML (e.g. style file, a row in a database) to be reused. This provides flexibility about what can be considered a reusable UI asset.

XInclude is a good mechanism for modularizing large specifications. It is also advisable if the reuse process is based upon a library of model specifications. However, if a reused asset changed those specifications that merged it will have to be updated too (fusion problem).

MBUIDE architectures that are based on a run-time facilitate external reuse. In particular, sharing run-time repositories and getting the run-time involved in the transparent location of the objects have brought us the following benefits: (a) overcoming the fusion problem, (b) flexibility about when and what to reuse (specifications can be reused at development time but UI objects can be also reused at execution time), and (c) the asset location is more comfortable since the developer does not need to know the full path of the object which facilitates the reuse implementation. We believe that a good context for using this technique would be the UI development in software products lines.

Regarding the management of reuse between applications, a centralized approach is advisable when multiple applications exhibit high commonality in the elements of some UI model and there is a single management domain. A federated approach is advisable when an application comprises elements distributed along various management domains (e.g. per model or other criteria).

8.4. *Open issues*

This work aimed to be a first step in the field of reuse analysis in MBUID. Consequently, a number of open issues can be highlighted for future research, such as the following:

- Increase the body of knowledge in the MBUID context by (a) exploring reusability in models that are also central in MBUID such as Task or Dialog and (b) analyzing other reuse techniques such as those in Table 1 (O.O., component-based, patterns, etc.).
- Perform and analyze comprehensive industrial use cases with systematic reuse processes and more detailed cost estimation, assessing not only productivity but also improvement in software quality and maintenance.
- Analyze reusability in the W3C standards for the AUI (Vanderdonckt et al., 2014) and Task models (Paternò et al., 2014). These standards are expected to have substantial impact in the MBUID community since more assets common to different environments will be generated due to (a) wider adoption of standard models by future system and (b) the possibility to use the proposed meta-models and its interchange syntax to transform UI assets between different user interface development environments (e.g. UsiXML and Maria).

9. Conclusions

In this paper we have presented WAINE, a MBUIDE that supports the reuse of UI assets defined in its models and have explained how its reuse techniques can be used along the UI development process. Using this particular system we have developed various applications within a common context. Results show that reusing UI assets with flexible techniques can provide a significant benefit in a cost-consuming task. We achieved a large reduction of specifications in the Presentation model between 34% and 75% depending on the project. Based upon our experience, we conclude that:

- The structure of UI models and its components has a significant impact on reusability. UI description languages should consider assets composition and granularity to create useful reusable assets at any level of abstraction in future MBUIDEs.
- MBUIDEs with run-time architecture can benefit from sharing repositories between various projects. Central repositories enable sharing UI assets that are common to various applications facilitating its maintenance. A federated system also allows that UIs are distributed into domains of management.

Acknowledgments

This research is partially funded by the Ministry of Science and Innovation of Spain (Grant DPI2010-19154) and the Andalusian Regional Government (Grant TIC-6339). We also would like to thank the anonymous reviewers for their constructive and quality reviews.

Appendix A. ASL code of sample application

Listing 3. Extract of an ASL document.

```
<!-- User model example -->
<group gid="1" name="rol1" mainid="rol1_taks">
    <user uid="1" name="user" descr="A_sample_user"/>
</group>

<!-- Dialog model example -->
<main id="rol1_taks" caption="Select_an_option">
    <menu caption="Data_managing">
        <option caption="A_sample_UI" call="csample"/>
    </menu>
    <menu caption="Misc">
        <option caption="Waine_web_page_" url="http://waine.us.es"/>
        <option caption="About" call="cabout"/>
        <option caption="Logout" action="LOGOUT"/>
    </menu>
</main>

<!-- Presentation model from Figure 4 -->
<form id="fremark" source="Remark" caption="Remarks">
   <fields>
      <key source="pk"/>
      <string source="descr" caption="Description" len="30"/>
      <fkey source="fkpatient"/>
   </fields>
</form>

<form id="fappointment" source="Appointment" caption="Appointments">
   <fields>
      <key source="pk"/>
      <date source="adate" caption="Date"/>
      <string source="descr" caption="Description" len="30"/>
      <fkey source="fkpatient"/>
   </fields>
</form>

<form id="fpatient" source="patient" caption="Patient">
   <orderby>name</orderby>
   <fields>
      <key source="pk"/>
      <string source="name" caption="Name" len="40"/>
      <date   source="bdate" caption="Birth_date"/>
      <string source="address" caption="Address" len="40" canbenull="Y"/>
      <int source="phone" caption="Phone" len="13" canbenull="Y"/>
   </fields>
</form>

<container id="csample" type="relation">
   <param name="form_split" value="rows=20%,*"/>
   <param ord="1" name="formid" value="fpatient"/>
   <param ord="2" name="containerid" value="csample_n"/>
</container>

<container id="csample_n" type="split">
   <param name="form_split" value="cols=50%,*"/>
   <param ord="1" name="formid" value="fappointment"/>
   <param ord="1" name="form_layout" value="table"/>
   <param ord="2" name="formid" value="fremark"/>
   <param ord="2" name="form_layout" value="table"/>
</container>
```

References

Abrams, M., Helms, User interface markup language (uiml) specification version 3.1. Technical report, Oasis UIML TC, 2004.

Ahmed, S., Ashraf, G., 2007. Model-based user interface engineering with design patterns. J. Syst. Softw. 80 (8), 1408–1422.

Baldassarre, M.T., Bianchi, A., Caivano, D., Visaggio, G., 2005. An industrial case study on reuse oriented development. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005, ICSM'05. IEEE, Budapest, pp. 283–292.

Basili, V.R., Caldiera, G. Improve software quality by reusing knowledge and experience. Sloan Manag. Rev. 37, 1995, 55.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J., 2003. A unifying reference framework for multi-target user interfaces. Interact. Comput. 15 (3), 289–308.

Delgado, A., Estepa, A., Estepa, R., 2007. Waine: automatic generator of web based applications. In: Third International Conference on Web Information Systems and Technologies, pp. 226–233.

de Sousa, L.G., Leite, J.C., 2006. Using imml and xicl components to develop multi-device web-based user interfaces. In: Proceedings of VII Brazilian Symposium on Human Factors in Computing Systems, IHC '06. ACM, New York, NY, USA, pp. 138–147.

Fonseca, J.M.C., Calleros, J.M.G., Meixner, G., Paternò, F., Pullmann, J., Raggett, D., Schwabe, D., Vanderdonckt, J., 2010. Model-Based UI XG Final Report. Technical Report, W3C (5).

Frakes, W., Kang, K., 2005. Software reuse research: status and future. IEEE Trans. Softw. Eng. 31 (7), 529–536.

Frakes, W., Terry, C., 1996. Software reuse: metrics and models. ACM Comput. Surv. 28 (2), 415–435.

Gill, N.S., 2003. Reusability issues in component-based development. SIGSOFT Softw. Eng. Notes 28 (4) 4–4.

Guerrero-García, J., González-Calleros, J., Vanderdonckt, J., Muñoz Arteaga, J., 2009. A theoretical survey of user interface description languages: preliminary results. In: Latin American Web Congress, 2009, LA-WEB '09, pp. 36–43.

Hyatt, David and Goodger, Ben and Hickson, Ian and Waterson, Chris. XML user interface language (XUL) 1.0, Mozilla. org, 2001.

Kontio, J., 1996. A case study in applying a systematic method for cots selection. In: Proceedings of the 18th International Conference on Software Engineering, ICSE '96. IEEE Computer Society, Washington, DC, USA, pp. 201–209.

Marsh, Jonathan and Orchard, David and Veillard, Daniel, XML Inclusions (XInclude) Version 1.0, W3C Working Draft, 10, 2006.

Meixner, G., Calvary, G., Vanderdonckt, J., 2010. Introduction to Model-based User Interfaces. Technical Report. MBUI Working group (5).

Meixner, G., Paternò, F., Vanderdonckt, J., 2011. Past, present, and future of model-based user interface development. i-Com 10 (3), 2–11.

Meskens, J., Vermeulen, J., Luyten, K., Coninx, K., 2008. Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me. In: Proceedings of the Working Conference on Advanced Visual Interfaces, AVI '08. ACM, New York, NY, USA, pp. 233–240.

Mohagheghi, P., Conradi, R., 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empir. Softw. Eng. 12 (5), 471–516.

Molina, P.J., 2004. A review to model-based user interface development technology. In: MBUI: Proceedings of the First International Workshop on Making Model-based User Interface Design Practical: Usable and Open Methods and Tools, Funchal, Madeira, Portugal, January 13, 2004.

Mascena, J.C.C.P., de Almeida, E.S., de Lemos Meira, S.R., 2005. A comparative study on software reuse metrics and economic models from a traceability perspective. In: IRI-2005 IEEE International Conference on Information Reuse and Integration, 2005. IEEE, pp. 72–77.

Mori, G., Paternò, F., Santoro, C., 2003. Tool support for designing nomadic applications. In: Proceedings of the 8th International Conference on Intelligent User Interfaces, IUI '03. ACM, New York, NY, USA, pp. 141–148.

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. Hum. Interact. 16 (4), 19:1–19:30.

Paternò, F., Santoro, C., Davide Spano, L., Raggett, D., 2014. Mbui—task models. Technical Report. World Wide Web Consortium (W3C). ⟨http://www.w3.org/TR/task-models/⟩.

Pinheiro da Silva, P., 2001. User interface declarative models and development environments: a survey. In: Interactive Systems Design, Specification, and Verification, Lecture Notes in Computer Science, vol. 1946. Springer, Berlin, Heidelberg, 2001, pp. 207–226.

Radeke, F., Forbrig, P., Seffah, A., Sinnig, D., 2007. Pim tool: Support for pattern-driven and model-based ui development. In: Coninx, K., Luyten, K., Schneider, K. (Eds.), Task Models and Diagrams for Users Interface Design. Lecture Notes in Computer Science, vol. 4385. Springer, Berlin, Heidelberg, pp. 82–96.

Sinnig, D., Javahery, H., Forbrig, P., Seffah, A. Patterns and components for enhancing reusability and systematic ui development. In: Proceedings of HCI International, Las Vegas, USA, Citeseer, p. 9, 2005.

Sousa, L., Leite, J., 2005. Jacob, R., Limbourg, Q., Vanderdonckt, J. (Eds.), Computer-Aided Design of User Interfaces IV. Springer, Netherlands, pp. 247–258.

Trætteberg, H., 2008. Integrating dialog modeling and domain modeling—the case of diamodl and the eclipse modeling framework. J. UCS 14 (19), 3265–3278.

Vanderdonckt, J., Furtado, E., Furtado, V., Limbourg, Q., Silva, W., Rodrigues, D., Taddeo, L., 2003. Multi-model and multi-level development of user interfaces. In: Multiple User Interfaces, Cross-Platform Applications and Context-Aware Interfaces, pp. 193–216.

Vanderdonckt, J.M., Bodart, F., 1993. Encapsulating knowledge for intelligent automatic interaction objects selection. In: Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems. ACM, Amsterdam, pp. 424–429.

Vanderdonckt, J., Tesoriero, R., Mezhoudi, N., Motti, V., Beuvens, F., Melchior, J., 2014. Mbui—Abstract User Interface Models. Technical Report. World Wide Web Consortium (W3C). ⟨http://www.w3.org/TR/abstract-ui/⟩.

Viana, W., Andrade, R.M., 2008. Xmobile: a mb-uid environment for semi-automatic generation of adaptive applications for mobile devices. J. Syst. Softw. 81 (3), 382–394.