AUTOMATYCZNE GENEROWANIE GRAFICZNYCH INTERFEJSÓW UŻYTKOWNIKA

dr inż. Mariusz Trzaska (*mtrzaska@pjwstk.edu.pl*)

Polsko-Japońska Wyższa Szkoła Technik Komputerowych;

ul. Koszykowa 86, 02-008 Warszawa

Deklaratywne tworzenie Graficznych Interfejsów Użytkownika jest również znane jako generowanie oparte na modelu. Większość tego typu rozwiązań wymaga wykorzystania dedykowanych narzędzi oraz dość dużego zasobu wiedzy. Zaproponowane podejście jest inne. Zdecydowaliśmy się opracować rozwiązanie, które znacząco ułatwia tworzenie typowych interfejsów dla istniejących popularnych języków programowania. Wykorzystaliśmy system adnotacji występujący we współczesnych językach takich jak Java, czy C#. W efekcie, programista jest w stanie określić, dla których elementów modelu, powinien zostać wygenerowany interfejs użytkownika. W najprostszej postaci, wystarczy tylko oznaczyć inwarianty klasy dla których powinny być stworzone komponenty GUI. Przedstawione podejście umożliwia również bardziej wyrafinowane opisywanie poszczególnych elementów, włączając w to etykiety, podpowiedzi, kolejność elementów, czy różne kontrolki. Po wygenerowaniu formatki przez programistę, użytkownik aplikacji jest w stanie tworzyć nowe jednostki danych oraz edytować już istniejące. Prezentowane badania są wsparte przez działającą prototypową bibliotekę dla języka Java. Jest ona częścią większego frameworku realizującego zarządzanie obiektami znacząco zmniejszającego pracę programisty wykonywaną w celu stworzenia aplikacji.

1. WPROWADZENIE

Zgodnie z [1] *Model-Based Development (MBD)* jest rozwijającym się trendem w dziedzinie współczesnej inżynierii oprogramowania. Częścią tego trendu są Graficzne Interfejsy Użytkownika oparte na modelu (*Model-Based GUIs - MB-GUI*), co oznacza, że GUI jest w jakiś sposób tworzone przez komputer na podstawie modelu (bez bezpośredniego zaangażowania projektanta/programisty). W [2] autor omawia różne rodzaje wykorzystywanych modeli: model aplikacyjny (*Application Model - AM*), model zadaniowo-dialogowy (*Task/Dialog Model - TDM*), czy model konkretnej prezentacji (*Concrete Presentation Model - CPM*). Wszystkie one są niezbędne, jeżeli ktoś chciałby opisać całą aplikację, włączając w to logikę biznesową (zachowanie się systemu). Niestety, taki opis jest dość skomplikowany i czasochłonny, a to oznacza, że nie jest zbyt popularny w środowiskach komercyjnych.

X Krajowa Konferencja Inżynierii Oprogramowania 2008 (KKIO'2008). Szklarska Poręba, Polska, 08 - 11 września 2008. ISBN: 978-83-206-1703-0. pp. 125-134.

Ogólnie rzecz biorąc, współcześni twórcy oprogramowania, wykorzystują trzy główne podejścia do tworzenia graficznego interfejsu użytkownika:

- ręczne pisanie kodu źródłowego odwołującego się do odpowiednich bibliotek. W przypadku Javy może to być Swing [3] lub SWT [4]. Programiści C# korzystają z WinForms [5]. Najbardziej skomplikowana sytuacja jest w C++, gdzie zastosowanie konkretnej biblioteki jest zdeterminowane przez dialekt języka. Istnieje również wiele niezależnych rozwiązań, która są czasami przenaszalne na różne platformy. Najbardziej popularne to Qt [6], wxWidgets [7] oraz GTK+ [8];
- wykorzystanie wizualnego edytora, który umożliwia „narysowanie" GUI i wygenerowanie odpowiadającego mu kodu źródłowego. Jakość takich generatorów jest bardzo różna. Niektóre z nich korzystają z tzw. inżynierii wahadłowej (*round-trip*) odzwierciedlającej ręczne modyfikacje kodu źródłowego na wizualny projekt. Istnieją również rozwiązania działające tylko jako czyste generatory kodu. Wtedy ręczne modyfikacje są tracone po ponownym wygenerowaniu kodu dla wizualnego projektu;
- zastosowanie specjalnego, deklaratywnego podejścia włączając w to MG-GUI. Jego zaletą jest to, że projektant/programista skupia się na tym „co jest do zrobienia", a nie „jak to zrobić". Najnowsze, komercyjne technologie podążające tym tropem to propozycja MS dotycząca języka XAML (*Extensible Application Markup Language*). Poszczególne elementy GUI są definiowane w specjalnym języku programowania, a właściwie języku opisu.

Niestety, większość przedstawionych podejść do tworzenia GUI, wymaga dość znaczącego zaangażowania ze strony programisty. Cały czas, każdy element GUI odpowiadający jednostce danych musi być „przetwarzany" indywidualnie. Przedstawiony pomysł polegał na stworzeniu rozwiązania, które jest bardzo łatwe w użyciu, a zarazem użyteczne i nie wymaga dużego nakładu pracy ze strony programisty. Opisywana propozycja została zaimplementowana w postaci biblioteki *senseGUI* dla języka Java i może być wykorzystana w dowolnej aplikacji pracującej na tej platformie. Warto podkreślić, że stosując opisane podejście, możliwe jest stworzenie podobnego rozwiązania dla dowolnego języka wspierającego refleksję (np. C#).

Aby dokładnie zrozumieć zaproponowane podejście, pewne ogólne informacje są umieszczone w podrozdziale 2. Podrozdział 3 opisuje sposób wykorzystania rozwiązania, a podrozdział 4 podsumowuje efekty prac.

2. ISTNIEJĄCE ROZWIĄZANIA

Przeciętny użytkownik aplikacji komputerowej, wykorzystuje graficzny interfejs użytkownika jako:

- wejście dla danych w celu wypełnienia ich pewną zawartością. W tym celu, programista tworzy pewne widgety (np. pole tekstowe) i łączy je z danymi (np. nazwiskiem klienta w systemie). Gdy użytkownik wprowadzi odpowiednie informacje, dedykowana część programu, zapisuje je w modelu;
- wyjście, celem zaprezentowania informacji przechowywanych w modelu. W tym celu, programista pisze kod odczytujący odpowiedni fragment danych i wyświetla go w określonym komponencie GUI.

Powyższe potrzeby mogą być zaspokojone korzystając z jednego z powyżej opisanych sposobów. Ponieważ stworzony prototyp jest dedykowany dla Javy, w dalszej części rozdziału skupimy się na rozwiązaniach przeznaczonych właśnie dla tego języka. Następne paragrafy zawierają krótkie omówienie implementacji potrzeb związanych z wejściem/wyjściem korzystając z istniejących rozwiązań.

Najbardziej powszechnym sposobem tworzenia GUI jest wykorzystanie bibliotek dostarczanych razem z językiem. Większość graficznych interfejsów użytkownika dla Javy jest implementowana używając bibliotek Swing [3] lub SWT [4]. Spójrzmy na kod poniższej klasy języka Java:

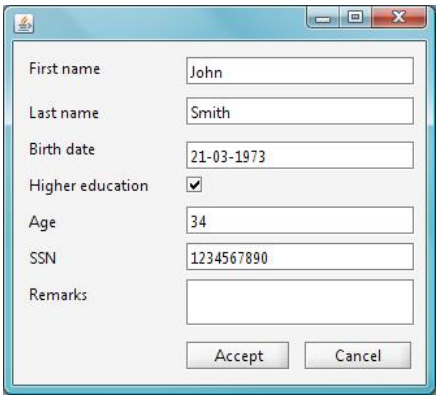
```
public class Person {
     private String firstName;
     private String lastName;
     private Date birthDate;
     private boolean higherEducation;
     private String remarks;
     private int SSN;
     private double annualIncome;

     public int getAge() {
       // [...]
     }
}
```

Aby opracować dla niego odpowiednie GUI, działające zgodnie z potrzebami wejścia/wyjścia, należy wykonać poniższe kroki:

- utworzyć pustą formatkę,
- dodać odpowiedni manager rozkładu,
- dla każdego atrybutu dodać widget, który zaprezentuje jego zawartość i umożliwi jego edycję,
- dla każdej metody dodać widget, który zaprezentuje wynik jej działania,
- dla każdego widgetu dodać odpowiednią etykietę,
- dla każdego widgetu dodać kod, który odczyta wartość określonego atrybutu i wstawi ją do widgetu,

– dodać przyciski kontrolne (Accept, cancel),
– dla przycisku “Accept” dodać kod, który odczyta zawartość widgetów, uaktualni odpowiedni fragment modelu oraz ukryje formatkę,
– dla przycisku “Cancel” dodać kod ukrywający formatkę.

Realizacja powyższych wymagań oznacza napisanie kilkudziesięciu linii kodu (7 atrybutów mnożone przez 5 do 10 linii na kontrolkę plus zarządzanie rozkładem, przyciskami kontrolnymi, itp.), które są do siebie dość podobne.



Rys. 1. Formatka dla klasy Person zaprojektowana przy użyciu edytora Jigloo

Aby zmniejszyć nakład pracy możemy skorzystać z edytora GUI. Jednym z nich jest Jigloo GUI Builder [9] przeznaczony dla środowiska Eclipse. Korzystając z tego rozwiązania projektant/programista jest w stanie graficznie zaprojektować formatkę odpowiednio rozmieszczając poszczególne widgety. Przykład dla klasy `Person` jest pokazany na rysunku 1. Dla tego formularza, edytor GUI wygenerował 105 linii kodu Javy. Ta liczba nie zawiera elementów niezbędnych do odczytu/zapisu wartości z/do modelu. Kod taki musi być stworzony ręcznie przez programistę. Mimo wszystko, jest to spore udogodnienie w porównaniu z ręcznym pisaniem całego kodu. Niestety programista musi spędzić trochę czasu rozmieszczając widgety, dodając kod obsługujący odczyt/zapis danych i zarządzający rozłożenia elementów.

Uważamy, że w przypadku typowych, biznesowych formularzy do wprowadzania/edycji danych, najbardziej obiecującym rozwiązaniem jest podejście deklaratywne. Powodem tego jest fakt, iż programista może skupić się na tym co chce osiągnąć, a nie jak to zrobić. Kolejną potencjalną zaletą jest możliwość przezroczystej pracy na różnych platformach (przenaszalność). W ogólnym przypadku, programista adnotuje kod źródłowy, a dedykowana biblioteka generuje odpowiedni kod zależny od platformy.

Systemy takie jak Teallach [10], Teresa [11], JUST-UI [12], SUPPLE [13] mają dość duże możliwości. Jednakże większość z nich pracuje z dedykowanymi językami definiującymi model (np. język wzorców w [12] lub UIML dla [11. Oznacza to,

że programista, który chce z nich skorzystać musi uczyć się czegoś nowego (i to dość skomplikowanego). Kolejną poważną wadą jest konieczność pracy na specjalnej platformie (np. Teresa dla [11]). Czasami ta nowa wiedza, którą trzeba przyswoić jest dużo bardziej skomplikowana niż pierwotny problem (GUI), który należało rozwiązać.

Zgodnie z naszą wiedzą, przedstawiony prototyp jest jednym z nielicznych rozwiązań umożliwiających tworzenie deklaratywnego interfejsu użytkownika na platformie Java.

3. MOŻLIWOŚCI BIBLIOTEKI senseGUI

Celem prac było stworzenie biblioteki, która automatycznie wygeneruje formularze GUI na podstawie zwykłych klas języka Java. Korzystając z tych formatek, użytkownik aplikacji będzie w stanie wprowadzić dane, uaktualnić istniejące informacje lub je po prostu przeglądać. Przykładowo, programista chce wykonać GUI (podobne do tego z rysunku 1) dla wcześniej przedstawionej klasy `Person`. Podstawowym założeniem było, iż cały proces odbędzie się bez żadnego zaangażowania ze strony programisty. Aby upewnić się, że zaproponowane rozwiązanie jest użyteczne, zdecydowaliśmy się zaimplementować działająca aplikację biznesową służącą do zarządzania wypożyczalnią wideo, wykonaną przy użyciu omawianej biblioteki.

W trakcie prac projektowych mieliśmy do rozwiązania trzy główne problemy:

- jak odczytać zawartość klasy (jej strukturę)?
- jaki widget powinien być użyty do poszczególnych rodzajów danych?
- jak połączyć poszczególne elementy GUI (widgety) z danymi?

Pierwszy problem rozwiązaliśmy przy pomocy techniki zwanej refleksją. Jest dostępna dla popularnych języków programowania (Java, C# i częściowo C++) i umożliwia odczytanie informacji o budowie klasy i tworzenie jej obiektów.

Odpowiedzi na resztę powyższych pytań są przedstawione w następnych podrozdziałach.

3.1. PODSTAWOWE MOŻLIWOŚCI

W czasie prowadzonych prac badawczych zdaliśmy sobie sprawę, że nie zawsze cała zawartość klasy powinna być edytowalna i/lub wyświetlana. Ponieważ nie jest możliwe automatyczne szacowanie, które elementy powinny mieć swoje odpowiedniki w GUI, byliśmy zmuszeni wprowadzić pewne znaczniki. Umożliwiają one dostosowanie generowanego interfejsu użytkownika do konkretnych potrzeb. Kwestią otwartą było w jaki sposób osiągnąć taką funkcjonalność, czyli połączyć programistę, bibliotekę i GUI? Mieliśmy do wyboru kilka możliwości, włączając w to pliki konfiguracyjne

(np. XML) lub przekazywanie parametrów w wywoływanych metodach. Ostatecznie, po przeanalizowaniu różnych rozwiązań, zdecydowaliśmy się wykorzystać adnotacje. Takie konstrukcje istnieją dla języka Java oraz C# i umożliwiają opisywanie klas oraz ich inwariantów. Staraliśmy się zaprojektować je maksymalnie prostymi w użyciu, ale w czasie implementacji przykładowego systemu, ich liczba wzrosła do 11. Na szczęście większość z nich ma domyślne wartości, których nie trzeba modyfikować. Wprowadziliśmy dwa podstawowe typy adnotacji: przeznaczone dla atrybutów (`GUIGenerateAttribute`) oraz metod (`GUIGenerateMethod`). Każdy z nich ma dodatkowe parametry (z domyślnymi wartościami):

- `label`. Opisuje etykietę znajdującą się przy widgecie. Jeżeli jest pusta (wartość domyślna) to zostanie wykorzystana nazwa atrybutu lub metody. Ten parametr był wymagany bo czasami musimy wprowadzić specjalne nazewnictwo (np. spację, czy polskie litery);
- `widgetClass`. Klasa widgetu, która będzie wykorzystana do edycji oraz wyświetlania danych. Domyślna wartość zakłada użycie pola tekstowego (`JTextBox`);
- `tooltip`. Krótki tekst wyświetlany po najechaniu kursorem na element.
- `getMethod`. Metoda wykorzystywana do odczytu wartości atrybutu. Domyślna wartość zawiera pusty ciąg i oznacza zastosowanie podejścia opartego na getterach oraz setterach;
- `setMethod`. Analogicznie jak w przypadku `getMethod`, ale dotyczy zapisu wartości;
- `showInFields`. Flaga określająca, czy ten element powinien być widoczny w formatce z polami;
- `showInTable`. Flaga określająca, czy ten element powinien być widoczny w widoku tabelarycznym;
- `showInSearch`. Flaga określająca, czy ten element powinien być widoczny w widoku służącym do wyszukiwania;
- `order`. Liczba określająca kolejność widgetu w formularzu;
- `readOnly`. Flaga definiująca, czy dane są dostępne w trybie tylko do odczytu (bez możliwości modyfikacji);
- `scaleWidget`. Określa, czy widget powinien zmieniać swój rozmiar w czasie modyfikacji wielkości formularza.

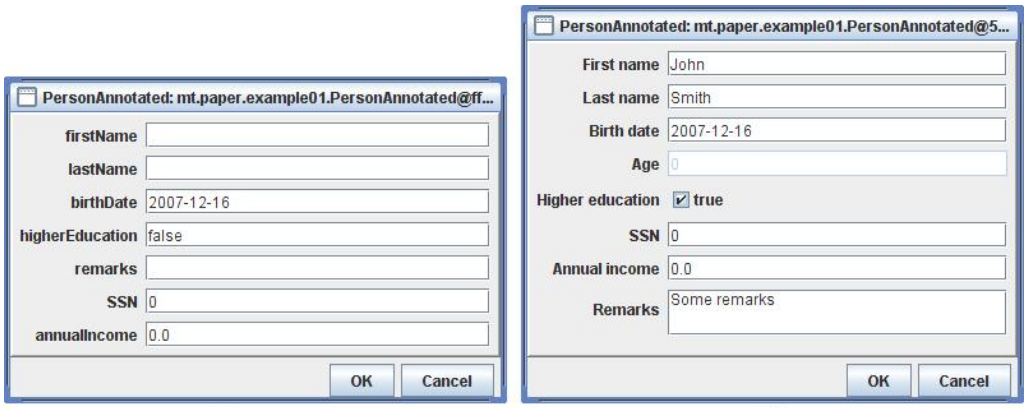
Opisywane rozwiązanie jest w stanie pracować z różnymi typami danych: liczbami, tekstem, wartościami Tak/Nie, datami typami wyliczeniowymi. Domyślna implementacja zakłada wykorzystanie pól tekstowych do wszystkich typów danych za wyjątkiem:

- wartości Tak/Nie – stosowany jest przełącznik (*check box*),
- typ wyliczeniowy (*enum*) jest przetwarzany w oparciu o specjalne pole wyboru zawierające wszystkie zdefiniowane w nim wartości (odczytane dzięki refleksji).

Poniższy listing zawiera kod przykładowej klasy `Person` udekorowany adnotacjami zdefiniowanymi w bibliotece. Warto zauważyć, że jedyne co musieliśmy zrobić to dopisać krótkie adnotacje dla wybranych atrybutów (korzystając z domyślnych wartości):

```
public class PersonAnnotated {
     @GUIGenerateAttribute
     private String firstName;
     @GUIGenerateAttribute
     private String lastName;
     @GUIGenerateAttribute
     private Date birthDate = new Date();
     @GUIGenerateAttribute
     private boolean higherEducation;
     @GUIGenerateAttribute
     private String remarks;
     @GUIGenerateAttribute
     private int SSN;
     @GUIGenerateAttribute
     private double annualIncome;
     @GUIGenerateMethod
     public int getAge() {
        // ...
     }
     // Standard getters/setters methods
}
```

W rezultacie wywołania pojedynczej metody z biblioteki, wygenerowana została formatka pokazana na rysunku 2a. Jest ona „połączona” z instancją klasy `Person`, dzięki czemu wprowadzone zmiany są automatycznie odzwierciedlane w modelu. Można zauważyć, że nie zawiera ona widgetu dla metody obliczającej wiek. Jest to zgodne z zaproponowanym podejściem, które zakłada, że przy domyślnym (bezparametrowym) użyciu adnotacji metody nie mają swoich odpowiedników na formatkach.



Rys. 2. Przykładowe formatki wygenerowane na podstawie adnotowanej klasy Java. Część (a) korzysta z domyślnych wartości, a część (b) ze zmodyfikowanych

Jak widać, takie proste udekorowanie kodu nie zawsze prowadzi do idealnych rezultatów. Z punktu widzenia użytkownika, można mieć zastrzeżenia do nazewnictwa poszczególnych elementów, kolejności w jakiej występują, czy zastosowanych widgetów. W takich sytuacjach warto wykorzystać odpowiednie parametry adnotacji, tak jak pokazano na poniższym listingu:

```
public class PersonAnnotated {
     @GUIGenerateAttribute(label = "First name", order = 1)
     private String firstName;
     @GUIGenerateAttribute(label = "Last name", order = 2)
     private String lastName;
     @GUIGenerateAttribute(label = "Birth date", order = 3)
     private Date birthDate = new Date();
     @GUIGenerateAttribute(label = "Higher education",
     widgetClass="mt.mas.GUI.CheckboxBoolean", order = 5)
     private boolean higherEducation;
     @GUIGenerateAttribute(label = "Remarks", order = 50,
     widgetClass="javax.swing.JTextArea",scaleWidget=false)
     private String remarks;
     @GUIGenerateAttribute(order = 6)
     private int SSN;
     @GUIGenerateAttribute(label= "Annual income", order=7)
     private double annualIncome;
     @GUIGenerateMethod(label="Age", showInFields = true,
     order = 4)
     public int getAge() {
             return 0;
```

```
        }
        // Standard getters/setters methods
    }
```

Efekt wygenerowania formularza na podstawie powyższego kodu pokazano na Rys. 2b. Warto zwrócić uwagę na poprawne nazwy, zmienioną kolejność elementów, czy wykorzystanie dedykowanego widgetu dla informacji o wykształceniu.

3.2. MOŻLIWOŚCI ZAAWANSOWANE

Niestety ze względu na ograniczony rozmiar niniejszego rozdziału nie jesteśmy w stanie w pełni opisać zaawansowanych możliwości biblioteki *senseGUI* działającej razem z pozostałymi elementami *senseObjects*. Warto tylko nadmienić, że programista może:

- zamiast korzystać z jednej „dużej" metody generującej całą formatkę, skorzystać z istniejących „mniejszych" metod umożliwiających dokładniejsze dostosowanie wyświetlanych elementów. Dzięki temu można wyświetlać zawartość atrybutów opisywanych za pomocą typów zdefiniowanych przez programistów (a nie tylko typów prostych);
- zarządzać powiązaniami pomiędzy obiektami. Stworzyliśmy odpowiednią funkcjonalność, która na podstawie adnotacji pozwala wyświetlać oraz zarządzać powiązanymi obiektami (asocjacje). Co więcej, można tworzyć kompozycje, asocjacje kwalifikowane, czy powiązania ograniczane przez {xor} lub {subset};
- wyświetlić wiele obiektów korzystając z widoku tabelarycznego wygenerowanego na podstawie adnotacji,
- automatycznie wyszukiwać obiekty na podstawie kryteriów podanych w specjalnej formatce,
- zarządzać ekstensją klasy.

Wykorzystanie powyżej opisanych funkcjonalności jest równie proste jak podstawowych możliwości opisanego rozwiązania. Dzięki połączeniu bibliotek (*senseGUI* + *senseObjects*) programista ma ułatwione tworzenie typowych, biznesowych aplikacji.

4. PODSUMOWANIE

Przedstawiliśmy rezultaty badań dotyczących automatycznego generowania graficznych interfejsów użytkownika dla aplikacji biznesowych. Wygenerowany formularz

jest oparty na prostych adnotacjach umieszczonych w kodzie źródłowym klas definiujących model danych. Programista wybiera, które inwarianty powinny mieć swoje odzwierciedlenie w GUI, a zaprezentowana biblioteka zajmuje się stworzeniem odpowiedniego interfejsu.

Uważamy, że wkład do stanu sztuki może być oceniony z dwóch punktów widzenia. Po pierwsze omawiana propozycja może działać z popularnymi technologiami (Java) i nie wymaga stosowania wyrafinowanych języków opisujących model. Po drugie, został stworzony działający prototyp, umożliwiający tworzenie aplikacji biznesowych.

Przyszłe prace skupią się na przeprowadzeniu formalnych testów użyteczności oraz wydajności. Chcielibyśmy także dalej rozwijać omawiane podejście, ponieważ uważamy, że deklaratywne (oparte na modelu) generowanie interfejsów użytkownika ma w sobie bardzo duży potencjał.

LITERATURA DO ROZDZIAŁU

[1] Basnyat S., Bastide R., Palanque P.: Extending the Boundaries of Model-Based Development to Account for Errors. MDDAUI '05 Model Driven Development of Advanced User Interfaces. 2005.

[2] da Silva P.: User interface declarative models and development environments: a survey. Proceedings of DSVIS 2000, 2000,pp. 207–226.

[3] Walrath K., Campione M., Huml A., Zakhour S.: The JFC Swing Tutorial (2nd Edition). ISBN 0201914670. Prentice Hall. 2004.

[4] Guojie J. L.: Professional Java Native Interfaces with SWT/JFace. ISBN: 978-0470094594. Wrox. 2005.

[5] Sells Ch., Weinhardt M.: Windows Forms 2.0 Programming. ISBN: 978-0-321-26796-2. Addison Wesley Professional. 2006.

[6] Thelin J.: Foundations of Qt Development. ISBN: 978-1-59059-831-3. Apress. 2007.

[7] Smart J., Hock K., Csomor S.: Cross-Platform GUI Programming with wxWidgets. ISBN: 978-0131473812 . Prentice Hall. 2005.

[8] Krause A.: Foundations of GTK+ Development. ISBN: 978-1590597934. Apress. 2007.

[9] Jigloo SWT/Swing GUI Builder: http://www.cloudgarden.com/jigloo/.

[10]Griffiths, Tonya, Barclay, Peter J, et al, Teallach: A model-based user interface development environment for object databases, Interacting with Computers,vol.1, 2001,pp.31-68.

[11]Mori G., Paterno F., Santoro C.: Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions, IEEE Transactions on Software Engineering, 30(8), 2004, pp.1-14.

[12]Molina P., Meliá S., Pastor O.: JUST-UI: A User Interface Specification Mode, in Proceedings of CADUI 2002, Valenciennes, France, 2002, pp.63-74.

[13]Gajos K., Weld D.: SUPPLE: Automatically Generating User Interfaces, in Proceedings of IUI'04, Funchal, Portugal,2004, pp.83-100.