Generation of user interface from characterized code

Jaroslav Kadlec
Faculty of Information Technology
Brno University of Technology
Bozetechova 2
612 66, Brno, CR
kadlecj@fit.vutbr.cz

Pavel Zemcik
Faculty of Information Technology
Brno University of Technology
Bozetechova 2
612 66, Brno, CR
zemcik@fit.vutbr.cz

ABSTRACT

Automatic generation of the user interface can simplify development of the computer applications. It can help in the development for various target platforms or in simpler testing and algorithm debugging as the user interface can be created for the actual code and platform taking into account many properties. Process of the automatic generation of the user interface can be supported by the data and code characterization. In this paper, an innovative approach using the data and code characterization is presented. The mechanisms and algorithms describing how the data and code characteristics are loaded, the way how objects are transformed into abstract and specific user interface elements, and the process of finalizing user interface is briefly described. As an example, simple media player is described in every step of the user interface generation process.

Keywords

code characterization, user interface, automatic generation

1. INTRODUCTION

Most of the applications in these days are created as one solution for one or more platforms. The applications portable on a multiple platforms are usually using some intermediate framework, such as Java or .NET Framework [Ell06a]. Many devices with the same or a different platform have very much varying properties and capabilities, so that the user interface created for an average system is not always the best option.

Development of the applications for more platforms simultaneously is a complex task and requires developers to have knowledge of all the required target platforms. Also applications developed for a single platform are composed from heterogeneous information about the algorithms, designs, and user interface which fact can render the design difficult. When the algorithms change, design and user interface code has to change too (to get control over the user interface elements).

For the above reasons, automatic generation of the user interface can simplify application development. The user interface elements can be generated to reflect algorithm changes. Cross-platform portability can benefit from the creation of user interface considering capabilities and properties of the executing system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2. PREVIOUS WORK

Number of the systems exists from 1980's that use various techniques for generating of user interface. A level of automation provided by these systems varies from the programming abstraction (e.g. UIML [Abr02a]) to design tools (e.g. ProcSee [PSe05a]), through the mixed systems requiring partial assistance from user interface designer (TERESA [Pat08a]). Such systems that provide some mechanisms to automatically generate user interface often use simple rule-based approach where every type is matched to the specific user interface element (e.g. UBI [UBI05a]). Some systems rely on the type-based declarative model of the information exchanged through the user interface called Abstract User Interface [Pat03a]. In many cases, a user interface was specified explicitly (e.g. UIML) or inferred from a code [Jel04a]. Some systems include additional information about a high level task or dialogue model (e.g. ConcurTaskTrees [Pat97a] or the task models [Bod94a]). Other systems generate the user interface using constraint satisfaction and optimization (e.g. Supple [Gaj08a]).

3. GENERATING USER INTERFACE

The approach presented here is based on data and code characterization [Kad07a] and it is combining rule-based concept for the interface abstraction from the characterized code and constraint satisfaction and optimization for choosing interface objects. General description of this approach was presented in [Kad06a].

The first step based on the approach is the code characterization. The characterized code is analyzed and the user interface is automatically created from the analysis, considering the properties of the platform, device, user preferences, and the user context. The next step is a rule-based creation of the user interface abstraction using the abstract interface objects. Then, the abstract interface objects are converted to the specific interface objects using the optimization and constraint satisfaction. Finally, the user interface is instantiated. Individual steps of the approach will be demonstrated on a "simple media player" example.

Data and Code Characterization

The first important step in the approach is the characterization of both the data and code. The data and code characterization is a process of annotation of the data and code with a special tags carrying important information for the future user interface generation process.

The characterization can be stored in a separate file or a directly in the source files (see Fig. 1) and compiled and linked to the library or assembly.

```
[DataType(Atomic)]
[DataDomain(Measurement, "Time" )]
[DataContinuity(Continuous)]
[DataTransience(Dynamic)]
[DataImportance(0)]
[CodeName("Time Line")]
[CodeDependence("IsMediafileOpened")]
[ParameterRange(0,0)]
public ulong CurrentPosition { get; set;}
```

Figure 1. Example of the characterized code in C# of a value representing position in the currently played media file.

Loading Code Characteristics

After the code and data characterization is done, the data and code characteristics should be loaded into a characterization tree. The characterization tree reflects well the structure of the data types and their properties and can be used for the creation of the abstract interface objects. A process of the creation of the characterization tree is shown in Fig. 2. First, an instance of the structure holding the characterization data is created (1). Second, attributes are parsed and stored in the structure (2).

```
LoadCharacteristics(tree, dataType) :
1.  instance = new Characteristics()
2.  ParseCharacteristics(instance, dataType);
3.  foreach(variable in dataType)
        a.  v = LoadCharacteristics(variable, instance);
        b.  instance.Data.Add(v);
4.  foreach(method in dataType)
        a.  m = LoadCharacteristics(method, instance);
        b.  m.Params = ParseParams(method);
        c.  instance.Code.Add(m);
5.  tree.Insert(instance);
```

Figure 2. Creation of a characterization tree.

Next, every variable is parsed recursively (3). If the variable has already been parsed, a reference to the node is saved. Similarly, for the methods, every method is parsed (4) including its parameters and stored in the tree (5). The resulting characterization tree of the media player is shown in the Fig. 3.

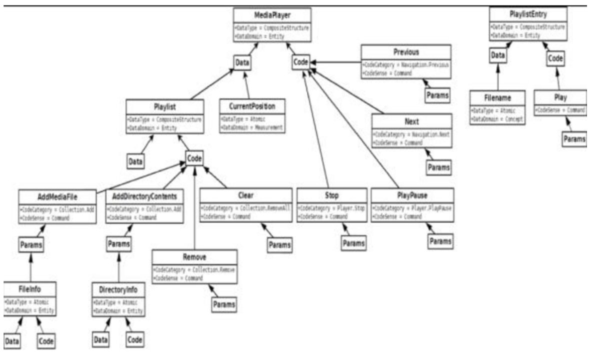
MediaPlayer
+DataType = CompositeStructure
+DataDomain = Entity
Data
Code
Previous
+CodeCategory = Navigation.Previous
+CodeSense = Command
Params
Playlist
+DataType = CompositeStructure
+DataDomain = Entity
CurrentPosition
+DataType = Atomic
+DataDomain = Measurement
Next
+CodeCategory = Navigation.Next
+CodeSense = Command
AddMediaFile
+CodeCategory = Collection.Add
AddDirectoryContents
Clear
+CodeCategory = Collection.RemoveAll
Stop
+CodeCategory = Player.Stop
PlayPause
+CodeCategory = Player.PlayPause
Remove
+CodeCategory = Collection.Remove
FileInfo
DirectoryInfo
PlaylistEntry
Filename
+DataDomain = Concept
Play

Figure 3. The characterization tree of the simple media player.

Creating Abstract Interface Objects

The Abstract Interface Objects (AIOs) are used to represent a user interface structure. They do not represent specific user interface elements but rather indicate what should be the data or code meaning in the terms of the user interface (the data structure can be e.g. represented as a container with a public data). The process of creation is rule-based with domain limitation and takes into consideration user context and preferences. The process of creation of AIOs is presented in the Fig.4. AIO database is loaded, including all the rules for every AIO. The characterization tree is parsed and for every node in the tree AIO is picked (1) and initialized (2). Initialization for every kind of AIO can be different.

To support a default behavior for commands, a smart template concept [Nic04a] is used (5.a) which groups commands of similar category together. For the methods that require attributes, a dialog AIO is created and filled with AIOs representing each parameter respectively (5.b).

```
Load AIO Database from repository.
CreateAIOs(char, prefs, context)
    1. selectedAIO = EvalRuleSet(char, prefs, context);
    2. selectedAIO.Initialize(char, prefs, context);
    3. char.AIO = selectedAIO;
    4. foreach(dch in char.Data)
        a. CreateAIOs(dch, prefs, context);
        b. selectedAIO.Add(dch.AIOs);
    5. foreach(cch in char.Code)
        a. if(cch has category && smart template exists)
            i.   cch.AIO = GetSmartTemplate(cch.category);
            ii.  cch.AIO.Link(cch);
        b. else
            i.   cch.AIO = EvalRuleSet(cch, prefs, context);
            ii.  if (cch.Params > 0)
            iii. dlg = CreateDialogAIO();
            iv.  foreach(param in cch.Params)
                     1. CreateAIOs(param, prefs, context);
        c. cch.AIO.Add(dlg);
```

Figure 4. Creation of AIO tree.

Fig. 5 demonstrates the resulting AIO tree created from the characterization tree. The main media player object is represented by a container, playlist as a collection, and seek-bar as a time measurement (both are sub-objects of media player class).

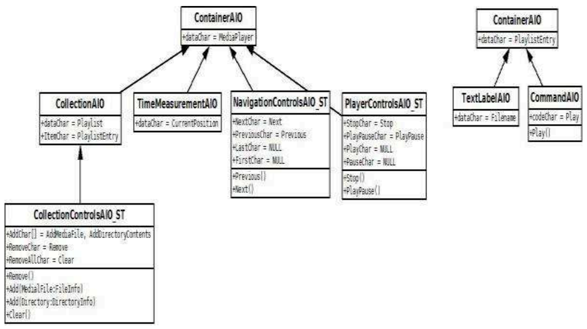

Figure 5. AIOs generated from the characterization tree.

The methods were linked together into three main AIOs thanks to smart template. The Playlist entry was placed separately during the collection initialization because it is used for the internal representation of a collection items and is not explicit part of the media player interface.

Creating Specific Interface Objects

The specific interface objects (CIOs) contain information about the specific user interface element that will be used in the final user interface. The AIO tree with a user interface structure is used to choose the best CIO for every data or code element. The presented process is generally enumeration of all the possible ways of choosing and inserting user interface elements. The best solution with the smallest effort needed for the interaction is chosen. This process is described in Fig. 6. The first step is evaluation of a cost function (1). The cost function evaluates the effort of the user in the interaction with current user interface objects in his current context and specified device. When the current cost is worse than the best solution found so far, conversion will not continue.

```
ConvertToCIOs(ChTree, AIO, Context, Device)
    1. If (CurrentCost(ChTree, Context, Device) >=
           BestCost) return;
    2. If (AllCIOsApplied())
        a. BestCost = Cost;
        b. BestCIOs = ChTree.CIOs;
        c. return
    3. CIOs = GetCIOs(AIO, Context, Device);
    4. foreach(CIO in CIOs)
        a. if(ApplyCIO(CIO, AIO, Device))
            i.   subAIOs = GetSubAIOs(AIO);
            ii.  SortByImportance(subAIOs);
            iii. foreach(subAIO in subAIOs)
                     1. ConvertToCIOs(ChTree, subAIO,
                        Context, Device);
    5. UndoLastCIO();

AIOsToCIOs()
    1. foreach(SubTree in ChTree)
    2. while(true)
        a. ConvertToCIOs(SubTree, SubTree.AIO, Context,
           Device)
        b. if (ConversionComplete(SubTree)) break;
        c. RegroupLowestImportanceContainer(SubTree);
```

Figure 6. Creation of CIO tree.

The second step is checking if all the AIOs were converted to the CIOs and saving solution (2). The third step enumerates all the CIOs available for the concrete AIO (3). Each of these CIOs is applied to the user interface without violating constraints. AIO conversion is repeated for sub AIOs recursively (4). AIOs with a higher importance are always placed first. Finally, the CIO is removed from the user interface because it can be replaced by other CIO in the 4th step of previous recursion.

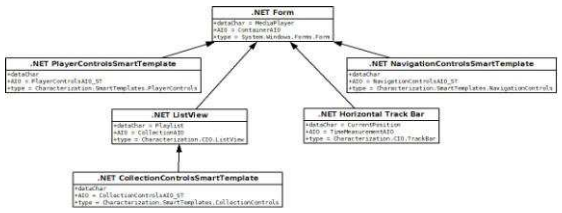
.NET Form
+dataChar = MediaPlayer
+AIO = ContainerAIO
+type = System.Windows.Forms.Form
.NET PlayerControlsSmartTemplate
+dataChar
+AIO = PlayerControlsAIO_ST
+type = Characterization.SmartTemplates.PlayerControls
.NET NavigationControlsSmartTemplate
+dataChar
+AIO = NavigationControlsAIO_ST
+type = Characterization.SmartTemplates.NavigationControls
.NET ListView
+dataChar = Playlist
+AIO = CollectionAIO
+type = Characterization.CIO.ListView
.NET Horizontal Track Bar
+dataChar = CurrentPosition
+AIO = TimeMeasurementAIO
+type = Characterization.CIO.TrackBar
.NET CollectionControlsSmartTemplate
+dataChar
+AIO = CollectionControlsAIO_ST
+type = Characterization.SmartTemplates.CollectionControls

Figure 7. Created CIOs from the AIO tree.

Fig.7. demonstrates the result of the conversion to the CIOs. Main class is represented by the Form (window), containing ListView for the playlist, track bar for the seek-bar and the smart templates for categorized commands.

Instantiating

Instantiation creates instances of CIOs and is responsible for generation and registration of events. The instance of every CIO is created so that instances have the same sub-objects as CIO nodes. Then, the parameters of the CIO are set to the instance. The CIOs representing a data register their dependencies on other objects, events for value changes of the data and the user interface instance. CIOs representing a code register their dependencies and implementing routines calls, generate events to show asterisks and code to show a dialog for input of the parameters if required.

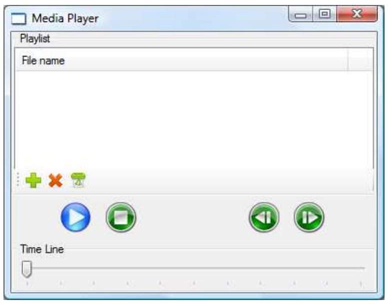

Figure 8. The final user interface.

Fig. 8 shows final user interface generated from CIO tree in Fig. 7. All CIOs were placed in the top-bottom and the left-right order representing highest to lowest importance.

4. CONCLUSION

The presented approach allows for automated creation of the user interfaces from the characterized code. It is based on rule-based creation of the abstract user interface and constraint satisfaction and optimization during creation of specific interface elements. It can be used to create a user interface for various modalities and user contexts. The rule-based approach can quickly reduce the set of the user interface elements for given modality and context while constraint satisfaction and optimization process can create interface with low interaction effort with a device and user capabilities in mind. It can be further extended to consider adaptation to usage. In situations, where the users are getting experienced in time, importance can be changed to reflect the most frequently used user interface elements. In situations, in which the user changes the environment, the current modality can be changed to enable more comfortable interaction.

5. ACKNOWLEDGEMENT

This work was supported by the Security-Oriented Research in Information Technology project sponsored by CEZ MŠMT under Grant No. MSM0021630528, and the Centre of computer graphics sponsored by MŠMT LC under Code No. LC06008.

6. REFERENCES

[Abr02a] Ali, M.F., Pérez-Quiñones, M.A., Abrams, M., Shell. E., Building Multi-Platform User Interfaces with UIML. CADUI 2002, France, pp. 255-266, 2002.

[Bod94a] Bodart, F., Hennebert, A.M., Leheureux, J.M., Provot, I., Vanderdonckt, J., A Model-Based Approach to Presentation: A Continuum from Task Analysis to Prototype, in Proc. of FIEW - DSVIS, pp: 25–39, 1994.

[Ell06a] Elliot, S. J2EE and .NET Applications: Creating Value Through Integrated Cross-Platform Management, IDC, 2006.

[Gaj08a] Gajos, K.Z., Wobbrock, J.O., Weld, D.S., Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces, in Proc. of 26th SIGCHI Conf. on HFCS, Italy, pp. 1257-1266, 2008.

[Jel04a] Jelinek, J., Slavik, P., GUI Generation from Annotated Source Code, in Proceedings of Tamodia'04, Prague, Czech Republic, 2004.

[Kad06a] Kadlec, J. Steps In Automated User Interface Generation, in Proc. of SCCG 2006, Bratislava, pp. 25-28, 2006.

[Kad07a] Kadlec, J., Code Characterization for Automatic User Interface Generation, In: Innovations and Advanced Techniques in CISSE, Dordrecht, NL, Springer, s. 255-260, 2007.

[Nic04a] Nichols, J., Myers, B.A., Litwack, K., Improving Automatic User Interface Generation with Smart Templates, In Intelligent User Interfaces, Funchal, Portugal, pp. 286-288, 2004.

[Pat03a] Paternò, F., Santoro, C. A Unified Method for Designing Interactive Systems Adaptable to Mobile and Stationary Platforms, Interacting with Computers 15, Elsevier, pp. 349-366, 2003.

[Pat08a] Paterno, F., Santoro, C., Mantyjarvi, J., Mori, G., Sansone. S., Authoring pervasive multimodal user interfaces. Int. J. Web Eng. Technol., Vol. 4, No. 2, pp. 235–261, 2008.

[Pat97a] Paterno, F., Mancini, C., Meniconi, S., ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models", in Proc. of the IFIP TC13 Int. Conf. on HCI, Sydney, pp. 362-369, 1997.

[PSe05a] Randem, H.O., Jokstad, H., Linden, T., Kvilesjo, H., Rekvin, S., Hornæs, A., ProcSee - The Picasso Successor, Enlarged Halden Programme Group Meeting, Lillehammer, Norway, 2005.

[UBI05a] Nylander, S., Bylund, M., Waern, A. The Ubiquitous Interactor - Device Independent Access to Mobile Services, CADUI 2004, Isle of Madeira, pp. 271-282, 2005.