Visual Interfaces Design Simplification through Components Reuse

Javier Rodeiro-Iglesias and Pedro M. Teixeira-Faria

School of Informatics Engineering, University of Vigo, Spain
jrodeiro@uvigo.es
School of Technology and Management - Polytechnic Institute of Viana do Castelo, Portugal
pfaria@estg.ipvc.pt

Abstract. One way to simplify a visual interface creating process is to give to the interface designer the ability of reusing pre-built visual components representations. In order to avoid premature commitment to specific presentations, and leaves open the prospect of alternative visual presentations for different environments, abstract interaction objects (AIOs) can be used. One of these AIOs is the *complex component*, which is a component representation having similarity properties with the object-oriented paradigm. This type of component embraces the reuse concept at semantic and functional levels, which contributes to reduce the complexity in the graphical user interface design process. Further advantages of using *complex components* are the possibility of visual and functional customization of these components, which greatly improves the versatility of them when compared with a *widget*.

Keywords: Abstract Interaction Objects, Complex Components, Visual User Interface Components Reuse.

1 Introduction

Most of the visual interfaces are created for the user to interact with them, using interactive visual components. Much work in the field of interactive graphics involves describing an interface in terms of a collection of *abstract interaction objects* (*AIOs*) [3][6][12]. An AIO represents a data structure of a user interface object without any graphical representation and independent of any implementation environment. The use of AIOs avoids premature commitment to specific presentations, and leaves open the prospect of alternative visual presentations for different environments. Therefore, in the interface designing process the selection of appropriate AIOs becomes necessary. In the study here described an AIO was selected: the *complex component (CC)*. The criteria considered to select it were by the fact that his representation supports visual appearance, topological composition and interaction [15]. It is also indicated that there exists a generic similarity between *(CC)* and *object-oriented paradigm* (*OOP*) which means that a relation between visual interface components (the *complex components*) and objects in OOP could be established. Knowing that OOP supports objects reusing (e.g. by association, aggregation or inheritance) [5] and

M. Kurosu (Ed.): Human-Computer Interaction, Part I, HCII 2013, LNCS 8004, pp. 441–450, 2013.

taking into account the comparable relation that exists between objects (in OOP) and *(CCs)*, the possibility of reusing *(CCs)* will be verified. The approach here proposed indicates the existence of a generic similarity between *(CCs)* and OOP [13] which means that a relation between these visual interface components and objects in OOP can be established (encapsulation, inheritance, polymorphism). For example, a *(CC)* has features that can be related with the *aggregation* concept existent in OOP, which differs from ordinary *object composition* in that it does not imply ownership. Thus, by eliminating one of the containers will not imply to eliminate the objects it contains (the same happens with *CCs*). Each container can be identified as a class, which keep a list of their child components, and allow adding, removing, or retrieving components amongst their children. In order to achieve this objective, we analyze the properties of component reuse at semantic and functional levels, based in one game visual interface prototype previously created [11].

1.1 Study Motivation

It is possible to establish the main scope of the study presented here. It is focused on the representation of self-contained visual interfaces based on the direct manipulation interaction style [10], supporting user freedom design features. The user interface designer can establish the shape, size, color, position, among other properties for each interface visual element. Thus, the user interface designer has the possibility to create a visual interface prototype based on visual components. Specifically, the contribution of this study is focused in verifying complexity reduction (simplification) in the visual interfaces design process, by using reuse features provided by *(CCs)* usage. The possibility of reusing *(CCs)* is of great importance since that contributes to simplify the interface design, which can be freely established by the interface designer. The visual elements to be used are independent of any platform or programming environment.

1.2 Defined Problem

In a previous study [15] the bases for characterizing a new AIO were established: this new AIO is called *(CC)*. Using this concept, an example of a game interface implementation was designed. And, after the interface has been designed, the idea of verifying the *(CC)* reuse features has emerged. It was decided to verify their reuse potential at both the semantic and functional levels. We understand the semantic level as the possibility to change *(CCs)* visual appearance, maintaining his functionality. And thus, allowing to use components on different platforms (e.g. to be possible to change the graphics of a game, while maintaining its functionality). The components reuse at functional level implies more profound changes in *(CCs)*, related with his functionality (e.g. more or less visual states and transitions between them). The problem that emerges is concerned with the validity of using these both reuse concepts. Thus, in order to verify that possibility, a relation with the OOP reuse concepts will be established, since it is a clearly stated and validated paradigm. Therefore, from an interface prototype designed using *(CCs)*, and assuming the

existence of a particular *(CC)* (with a specific visual appearance and behavior) which the designer wants to reuse in another interface, it will be verified if it can be done under considering two perspectives. The first one is semantic, by keeping the component behavior and changing the visual appearance. The other perspective implies to change the component behavior while keeping (or not) the original visual appearance. Considering these two perspectives, an analysis will be made in order to validate them focusing in the OOP reuse concepts.

2 Components Specifications

Usually, the term *reusability* is related with OOP technology and most of the times specifically related to *reusing code* [1]. Other related term is *inheritance reuse* which refers to using the inheritance concept in an application, in order to take advantage of the behavior implemented in existing classes. Other term is *component reuse* which refers to the use of prebuilt, fully encapsulated components, usually called *widgets* (*WIndows gaDGETS*). They are typically self-sufficient and encapsulate only one concept. Usually, the component reuse concept differs from code reuse in that we don't have access to the source code and it differs from *inheritance reuse* in that it doesn't use *subclassing* (new classes based in existing ones). Common examples of reusable software components are *Java Beans* and *ActiveX* components. There are several advantages in component reuse. First, it offers a greater scope of reusability than either code or inheritance reuse because components are self-sufficient (typically, we plug them in and they work). The main disadvantage of component reuse is that because components are small and encapsulate only one concept, we may need a large library of them to create an application (although when a component encapsulates one concept, it is a cohesive component).

2.1 User Interface Description Languages (XML-UIDL)

During the last decade, new user interface specification tools have emerged, with special focus on *User Interface Description Languages Based on XML* (*XML-UIDL*).

To specify user interfaces using XML [17] is considered to be one solution for the standardization and interoperability between applications [9][14] and is the main reason for the constant emergence of new XML-UIDLs. It is possible to observe (Figure 1) the release year of XML–Compliant languages first versions (drafts in some cases). Each of them comes up with a specific purpose and application. For example, one of those description languages is the *XForms* [16]. It separates the presentation from the data, keeping the principle of separation of concepts, allowing component reuse and device independence. However, despite XML supports reuse, and some of the (XML-UIDL) allow visual presentation reusing, these languages are not designed to support functional reuse. Thus, this type of interface specifications is not considered in our analysis related with components reuse, and thus another approach was taken.

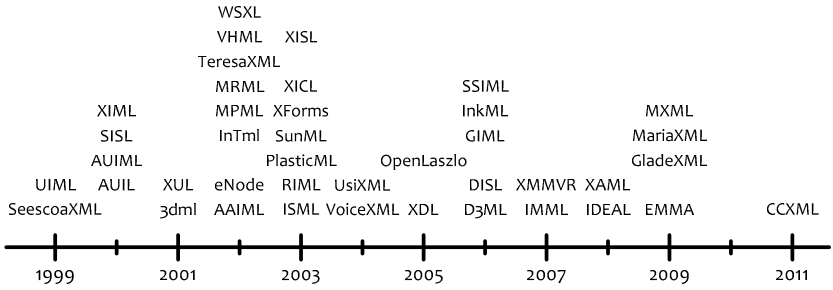

Fig. 1. XML-UIDL Evolution

2.2 Abstract Interaction Objects (AIOs)

A *Concrete Interaction Object* (CIO) represents any visible and manipulable user interface object that can be used to input/output information related to user's interactive task, and sometimes called widgets. These objects include some restrictions [2]:

- Lack of uniformity and standardization: concrete interaction objects induces a generalization problem as soon as a same object can be found in different physical environments with different names, different graphical presentations, but still with the same behavior;
- Absence of abstract representation: without such a representation, developers are submitted to specificities of several physical environments, designers are forced to not ignore low level details, and human factors experts are mainly focused on presentation aspects rather than behavioral aspects;
- Lack of compatibility with OO programming: in this programming paradigm most object classes' libraries encapsulate logically related classes with respect to inheritance relationship. Basic classes currently provide foundation classes, widget classes and graphical object classes. Any abstraction that is not compatible with OO dedicated mechanisms will be limited and useless;
- Difficulty of reusability: the reuse of existing objects leads to the acceptation of existing CIO's constraints which can be considered as insufficient under given circumstances. Creating new AIO from existing AIOs will be less hazardous as abstract properties (e.g., attributes) can be reused from one AIO to another.
- These shortcomings clearly motivate the need for an abstract interaction object AIO. Considering that, four AIOs have been analyzed: *interactor* [6][8][15], *abstract data view (ADV)* [4][15], *virtual interaction object* [12] and *complex component* [15]. The AIOs analysis considered here is focused in his visual appearance and interaction properties (Table 1). Concerning visual appearance, we verified that two of the AIOs don't support visual presentation and one of them doesn't consider visual states (interactor has states, but are not visual and are algebraically represented). Interaction refers to the bi-directional interaction from or to an *interaction object*. In general, three elements may interact with an interaction object: the

user, another *interaction object* and the *application*. After the four AIOs have been analyzed, considering several characteristics (Table 1) we decided to choose the *(CC)* to analyze its reuse properties. This choice took into consideration the *(CC)* be the AIO which agglutinated more features supported in part by the other AIOs. Basically, a *(CC)* is a component composed of other components (*simple* or/and *complex*) which interact with each other through its *self* and *delegate events/actions* working toward a common goal (e.g. a *toolbar* allows a user to select a specific tool to perform some task at a given time) [15]. The components follow a hierarchical topological structure and so each one can be contained within others. Thus, an analysis on semantic and functionality perspectives of (*CCs*) reuse is presented.

Table 1. AIOs comparison

	interactor	***abstract data view (ADV)***	***virtual interaction object***	***complex component***
Visual Presentation			×	×
Visual States		×	×	×
Input from User	×	×	×	×
Output to User	×	×	×	×
Input from the Application	×			×
Output to the Application	×			×
Input from Other Components	×		×	×
Output to Other Components	×		×	×

3 Semantic Perspective of Components Reuse

The visual interface of a game was designed and when the user looks at the interface he has at his disposal two perfectly distinct groups of visual elements (which correspond to three balls and three sport fields). The interface functionality was implemented using *(CCs)* at two abstraction levels: in one of them, 6 *(CCs)* were used (each one corresponding to one ball or one field) and in the other abstraction level, 2 *(CCs)* were used (one corresponding to a group of balls and the other corresponding to a group of fields). As previously mentioned, a characteristic resulting from using *(CCs)* to represent an user interface is related to the ease of components reuse. In a first perspective to that, the interface designer can create a new user interface maintaining its functionality. A new visual interface is immediately obtainable, due to the fact that *(CC)* concept to consider components reusability in his characteristics. If the interface designer wants to reuse a *(CC)* in another interface, keeping the functionality but with a different visual appearance, he can do it. This perspective is focused in drawing a new game interface by simply changing the component visual states, while still maintaining its functionality. Instead of the user (in this case a child)

has to relate balls with sport fields, he could for e.g. to relate objects with colors or sport shoes with balls. In this semantic *(CCs)* reuse perspective, the designer only has to be care with changing visual presentation attributes.

4 Functionality Perspective of Components Reuse

Another perspective of components reuse can be analyzed considering changes in the functionality of the *(CCs)* used. In a first approach, the changes in the components functionality are related with the number of components contained inside a *(CC)* (number of visual elements to be used) (e.g. instead of using three balls and three fields, a reduction or an increase in the number of available components could be tested, maintaining the components functionality). With respect to this approach it will be important to verify and to assess the changes occurring in parameters associated with the new visual interface (events, states, visual transitions) according to the reduction or to the increase in the number of used components.

In OOP a *class* is defined as a base structure used to create instances of it (objects). We can identify a *(CC)* as an interface component (with visual appearance, composition properties and supporting user interaction) which can be compared to an OOP class. A general comparison was previously made [15]. However, this part of the study will be focused on verifying *(CCs)* reuse features comparing them with the OOP reuse provided by the concepts of *association*, *aggregation* and *inheritance* [7].

4.1 Components Creation by Association and Aggregation Approach

An association represents a relationship between classes, and gives the common semantics and structure for many types of "connections" between objects. Associations are the mechanism that allows objects to communicate to each other through messages. Analogously, the communication between *(CCs)* associated with each other is performed by using *delegate events/actions.*

Class Association. Each ball *(CC)* used to design the game previously referred can be compared with a class with 3 possible visual states (*normal*, *selected* and *correct*) and 3 methods responsible for changing those states (visual transitions). Also each field *(CC)* can be identified as having features like a class with 2 visual states (*normal* and *correct*) and one method (visual transition). The structure of a ball and a field is represented on Figure 2.

On Figure 2 we verify that each ball may receive 2 user events and triggers 1 *delegate event* (on other component). It is also verifiable that there are internal transitions between the states of the *(CC)* limited by restrictions. In the case of the field, it receives 1 user event and triggers 1 *delegate event*. It has an internal transition between the 2 states whose trigger is dependent on a restriction. Relating the *(CC)* concept with the class concept, 3 *CC_Ball* class instances and 3 *CC_Field* class instances need to be created to implement the referred game. The relation between these balls and fields classes can be established by the OOP *association*, which

defines a relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf. In this case we verify the similitude with OOP method invocation, by the action performed by a *delegate event* triggered from a *(CC)*.

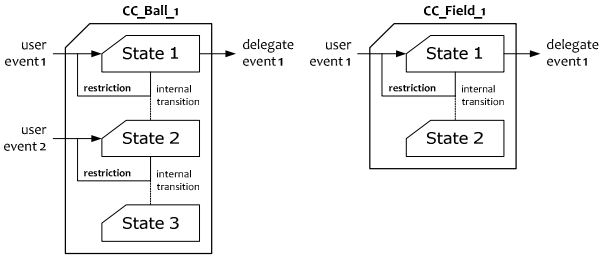

Fig. 2. Structure of a ball (left) and a field (right) used in the game interface

Class Aggregation. Aggregations are a special type of associations in which the participating classes don't have an equal status, but make a "whole-part" relationship. A *(CC)* also has features that can be related with the *aggregation* concept existent in OOP. The *CC_Balls* and the *CC_Fields* act as containers of 3 balls and 3 fields, respectively (Figure 3).

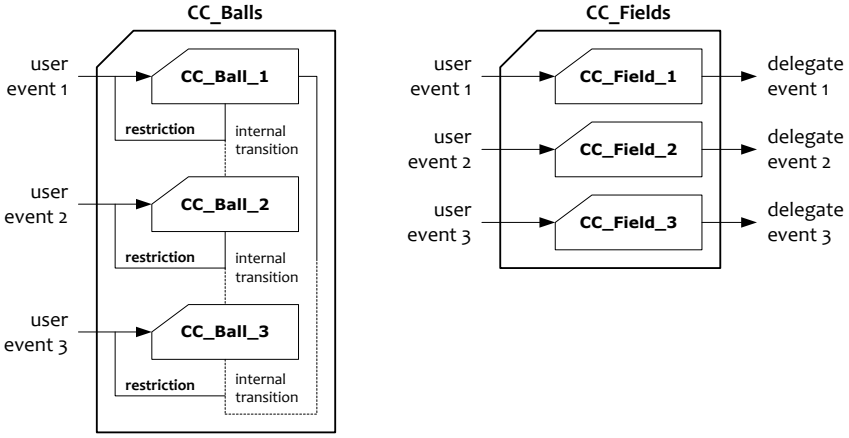

Fig. 3. Structure of the *CC_Balls* and *CC_Fields* containers used in the game interface

However, *aggregation* differs from ordinary *composition* in the perspective that it does not imply ownership. Thus, by eliminating one of the containers will not imply to eliminate the objects it contains. Each container could be identified as a class, which keep a list of their child components, and allow adding, removing, or retrieving components amongst their children.

4.2 Components Creation by Inheritance Approach

After analyzing *(CCs)* reuse approach by using the components, maintaining his original states and visual transitions, it seems to be adequate to analyze components reuse through another perspective, in which a *(CC)* is modified in order to contain more states and visual transitions. Thus, the relation of *(CCs)* design with the inheritance concept OOP will be verified.

Complex Component Application Domain Change. In order to expand and to verify the level of usage of a *(CC)* it was decided to change the application domain. Thus, *CC_Balls* has been chosen to be reused as a toolbar visual component. As previously mentioned, it is possible to reuse a *(CC)* by simply changing its visual appearance and hence his semantic. However, beyond the domain change it is intended to change the number of states and visual transitions of the *(CCs)* that compose the chosen container *CC_Balls (CC)*. In this way, the following changes were decided to perform:

- Increase the number of visual states: it is intended that each *(CC)* inside *CC_Balls* has one more visual state (e.g. each ball has three visual states and it is intended that each tool in the toolbar has four visual states);
- Increase the number of transitions between visual states: each *CC_Ball (CC)* inside *CC_Balls* contains three possible visual transitions between the three visual states. It is intended that each tool in the toolbar has five possible visual transitions between the four visual states.

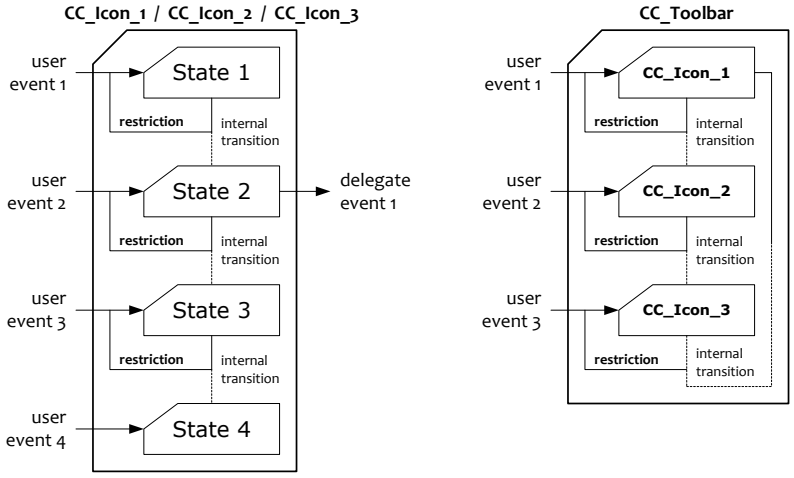

Fig. 4. The 3 icons (3 equal instances on the left) and the toolbar (on the right) represented as *complex components*

On Figure 4 are indicated the structures of each of the 3 tools (*CC_Icon_1*, *CC_Icon_2*, *CC_Icon_3*) which are inside (*CC_Toolbar*). Keeping in mind the possibility of reusing *(CCs)*, comparing it with reuse in OOP, we verify the similitude with the inheritance concept. Basically, considering a *CC_Balls (CC)* (which contains

3 balls) represented as a class, we may reuse it as a toolbar (with 3 tools) through the creation of a class by inheritance and extending it to support a new visual state and to redefine the existent transitions (by keeping some, eliminating and creating other). We verify a close relation between *(CCs)* and OOP. One of the important advantages of OOP is that it promotes reuse. Therefore, if we use *(CCs)* to design visual interfaces, similarly we can do component reuse of those components.

5 Conclusions

It is possible to design visual interfaces through various existent component specifications. One way to optimize the design process is through the reuse of components. During the last decade there has been a huge growth in the number of user interface description languages which uses XML as support language (*XML-UIDL*). However, those specifications need to be connected with high-level components provided by toolkits and usually referred as *widgets*. The use of those *widgets* limits the customization options available and because of that it limits the potential of reuse of this type of visual components. Additionally, even in spite of an increasing number of XML-UIDL enabling visual presentation reuse, such XML–Compliant Languages do not allow components functional reuse. Thus, we decided to increase the level of abstraction in the components specification by using AIOs. From those which were verified, the one which best supports features related with visual presentation, topological composition and component interaction is the *(CC)*. Thus, we sought to determine whether this type of component supports reuse. It was possible to verify the existence of a similitude between *(CCs)* features and OOP characteristics, considering in particular the reuse. This reuse characteristic can be applied by a *(CC)* under a semantic or a functional perspective. It contributes to reduce the complexity in a graphical user interface design process by reducing the number of components created and used. However, two envisaged limitations concerned with *(CCs)* reuse are:

- Identification of the issues that is necessary to change in *(CCs)* characterization in order to enable reusability;
- Classify/distinguish a new *(CC)* created by reusability.

In spite of those limitations, this advantage of being possible to reuse components simplifies the interface specification process also by being possible to specify (individually) each component and then, at the global interface level, only be necessary to specify the part that is not done yet with the *(CCs)*. Therefore, the obtained results confirm the hypothesis of being possible to simplify a graphical user interface design through the use of *(CCs)* which supports reusability. Thus, further advantages of using *(CCs)* are the possibility of visual and functional customization of these components, which greatly improves the versatility of a *(CC)*, when compared with a *widget*.

Acknowledgments. This work was supported by:

1. Grant SFRH/PROTEC/49496/2009 of MCTES – Ministério da Ciência, Tecnologia e Ensino Superior (Portugal).
2. Project TIN2009-14103-C03-03 of Ministerio de Ciencia e Innovación (Spain)
3. Project 10DPI305002PR of Xunta de Galicia (Spain).

References

1. Ambler, S.: A realistic look at object-oriented reuse. Software Development 6(1), 30–38 (1998)
2. Bodart, F., Vanderdonckt, J.: Widget Standardization through Abstract Interaction Objects. In: Proceedings of 1st International Conference on Applied Ergonomics, pp. 300–305. Springer, Istanbul (1996)
3. Carr, D.: Specification of Interface Interaction Objects. In: CHI 1994 – ACM Conference on Human Factors in Computer Systems, pp. 372–378 (1994)
4. Cowan, D., Lucena, C.: Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. IEEE Transactions on Software Engineering 21, 229–243 (1995)
5. De Champeaux, D.: Object-Oriented Analysis and Top-Down Software Development. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 360–376. Springer, Heidelberg (1991)
6. Duke, D., Harrison, M.: Abstract interaction objects. Computer Graphics Forum 12(3), 25–36 (1993)
7. Eck, D.: Introduction to Programming Using Java, 6th edn. (2011)
8. Faconti, G., Paternó, F.: An approach to the formal specification of the components of an interaction. In: Vandoni, C., Duce, D. (eds.) Eurographics 1990, pp. 481–494. North-Holland (1990)
9. Guerrero-Garcia, J., González-Calleros, J., Vanderdonckt, J., Muñoz-Arteaga, J.: A Theoretical Survey of User Interface Description Languages: Preliminary Results. In: Latin American Web Congress, pp. 36–43 (2009), doi:10.1109/LA-WEB.2009.40
10. Hutchins, E., Hollan, J., Norman, D.: Direct Manipulation Interfaces, vol. 1, pp. 311–338. Lawrence Erlbaum Associates, Inc. (1985)
11. Rodeiro-Iglesias, J., Teixeira-Faria, P.M.: User Interface Representation Using Simple Components. In: Jacko, J.A. (ed.) Human-Computer Interaction, Part I, HCII 2011. LNCS, vol. 6761, pp. 278–287. Springer, Heidelberg (2011)
12. Savidis, A.: Supporting Virtual Interaction Objects with Polymorphic Platform Bindings in a User Interface Programming Language. In: Guelfi, N. (ed.) RISE 2004. LNCS, vol. 3475, pp. 11–22. Springer, Heidelberg (2005)
13. Schlungbaum, E., Elwert, T.: Dialogue Graphs - A Formal and Visual Specification Technique for Dialogue Modelling. In: BCS-FACS Workshop on Formal Aspects of the Human Computer Interface. Sheffield Hallam University, Springer (1996)
14. Souchon, N., Vanderdonckt, J.: A Review of XML-compliant User Interface Description Languages. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) DSV-IS 2003. LNCS, vol. 2844, pp. 377–391. Springer, Heidelberg (2003)
15. Teixeira-Faria, P.M., Rodeiro-Iglesias, J.: Complex Components Abstraction in Graphical User Interfaces. In: Jacko, J.A. (ed.) Human-Computer Interaction, Part I, HCII 2011. LNCS, vol. 6761, pp. 309–318. Springer, Heidelberg (2011)
16. W3C, XForms 1.0: The neXt generation of web FORMS, W3C Recommendation (October 14, 2003), `http://www.w3.org/TR/2003/REC-xforms-20031014/`
17. W3C Recommendation: XML, XML 1.0 (2008), `http://www.w3.org/TR/REC-xml/`