A Declarative DSL Approach to UI Specification - Making UI's Programming Language Independent

Sofie Goderis * Dirk Deridder †
sgoderis@vub.ac.be dderidde@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium

Keywords with respect to the workshop

Meta-Programming, Customization, Configuration, Adaptation, Composition, Inheritance

1 Introduction

The problem of separation of concerns exists in many flavours and is widely acknowledged in the software development community [HL95]. Currently a large number of researchers are trying to find innovative ways to tackle this problem. The most well-known research fields that focus on this are aspect-oriented programming [KLM+97], component based programming [Szy98], ... The techniques proposed by these researchers depend on the type of concern they focus on (examples of this are transaction management, logging, synchronization, ...).

In our research we focus on the concern of User Interfaces and application interactions. In practice the user interface concerns are seldom separated from the application concerns. A clean separation would nevertheless facilitate UI development and evolution. Nowadays the UI representation (i.e. drawing windows, text-boxes, buttons, ...) can easily be separated from everything else. After drawing the UI, in general some stubs are generated which then have to be filled in with details and interactions. Evolving these UI's is quite problematic since the evolving UI can no longer be drawn (in order not to lose the hand-made fill-ins) and requires manual adaptation.

Figure 1 depicts the idea of separating the UI from its underlying application. One should keep in mind that both interact externally with each other and internally with themselves. When separating concerns it will therefore be important to focus on factoring out these interactions since they are the main cause for the code entanglement. In our work we focus on the interactions between the UI and the application (figure 1 number 3), and within the UI itself (figure 1 number 2), as well on the different layers of abstraction within the GUI box (figure 1 number 1).

*Author is financed with a doctoral grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT)

†This research is partially performed in the context of the e-VRT Advanced Media project (funded by the Flemish Government) which consists of a joint collaboration between VRT, VUB, UG, and IMEC.

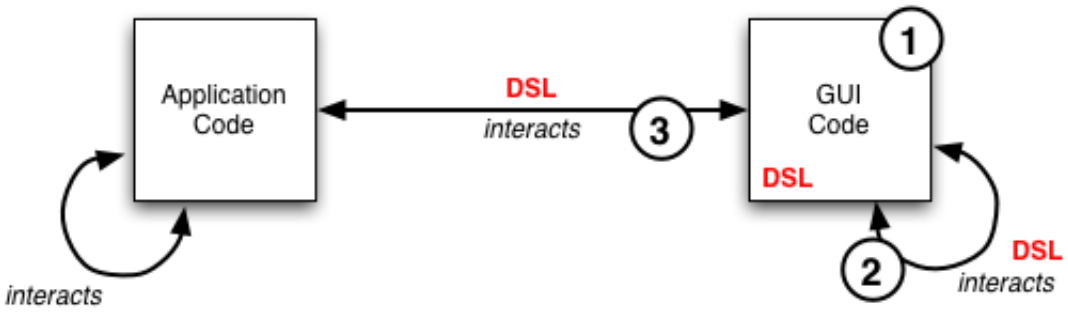

Fig. 1: Separating GUI and Application Interactions

Once the separation between UI and application is achieved, different examples can be thought of where separation is beneficial. For instance, deploying the UI to another platform without having to rebuild completely, or adapting the interface to different kinds of users.

In order to solve this problem we will express high level UI components (i.e. the GUI components in figure 1 (1 and 2)) by means of declarative meta-programming. This denotes that a user interface is described declaratively by means of rules and facts and later on is composed through a problem solver. Different concerns, as well as different UI abstraction layers, will lead to different sets of rules. The problem solver will combine all of these in order to get to the user interface that is wished for. Consequently we provide a declarative user interface framework as a domain specific language to program user interface concerns and their interactions with the application code.

2 User Interface Framework

In order to better understand the discussion topics in section 3, we give a broad overview of the User Interface Framework we envision (figure 2). The numbers in the figure correspond to the numbers in figure 1. Different sets of specifications (facts and rules) are created for different layers of abstraction, going from a very abstract user interface specification (figure 2 part **c**) towards a low-level (figure 2 part **b**) and specific (figure 2 part **a**) specification. There will be rule sets dedicated to the UI building blocks, to layout mechanisms, to interaction between UI components and to the interactivity between UI and application.

As a Declarative Meta Programming language we use SOUL (Smalltalk Open Unification Language) [Wuy01]. SOUL is an interpreter for Prolog that runs on top of a Smalltalk implementation. Besides allowing Prolog programmers to write 'ordinary' Prolog, SOUL enables the construction of Prolog programs to reason about Smalltalk code. Amongst others this enables declarative reasoning about the structure of object-oriented programs and declarative code generation.

The SOUL Problem Solver will use the rules and facts in order to come up with a solution. We anticipate that we will need to extend this standard problem solver with a constraint problem solver. In what follows we will go into the following components from figure 2 : UI building block rules, UI interaction rules and application interactivity rules, and the layout strategy rules.

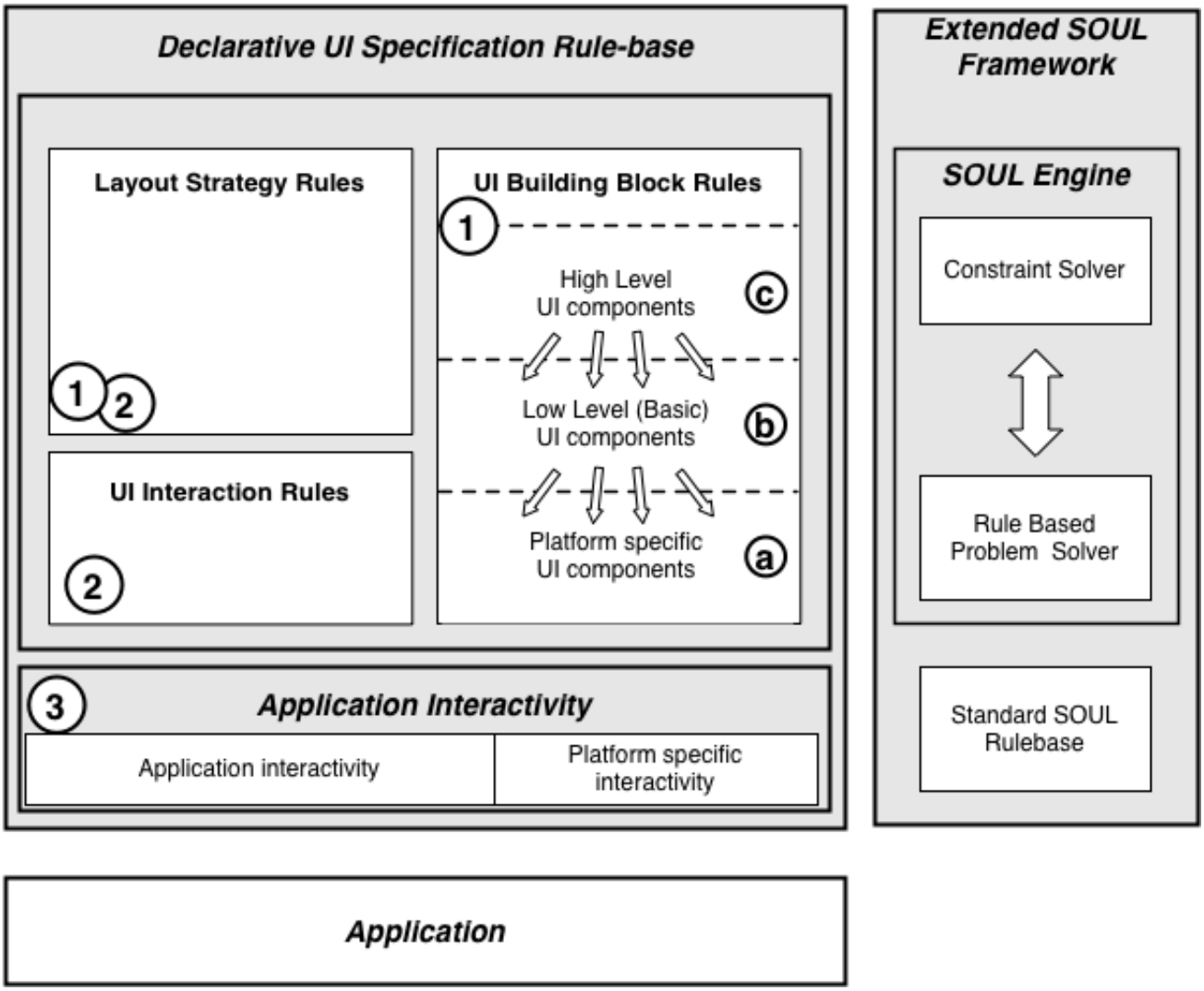

Fig. 2: Declarative UI Framework

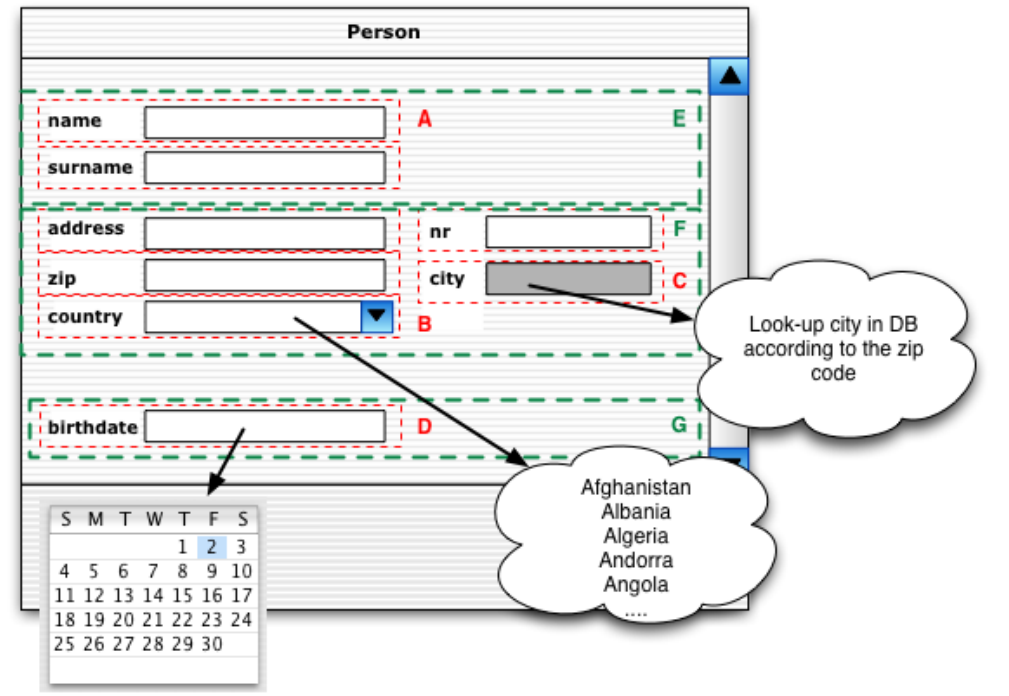

Fig. 3: Address book example

2.1 UI Building Blocks

In this section we will focus on the UI Building Blocks (figure 1 number 1). Let us consider a simple address book application. The UI for adding a person is shown in figure 3. The UI is composed of different basic entities which we can classify as labels, input-fields, drop-down box and text-field. The description for these components will be part of the platform specific UI building blocks (figure 2 part **a**).

However, the UI could be seen in a more abstract way (as depicted in figure 3 by boxes A, B, C and D) with a number of different input components and a computed value component. A first input component (A) is of the type *simple* and has a label *name*. Another input component (B) is of the type *choice* and has a label *country* and choices *Belgium, The Netherlands, France, UK, ...* . The computed value component (C) has a label *city* and certain *behavior* that computes this value based on the *zip code*. These components belong to part **b** in figure 2.

Figure 4 gives the SOUL rule for a simple input component that makes use of the smalltalk platform specific components *label* and *inputfield*. The inputComponent rule is used in order to create the *name* label.

Abstraction can be taken a step further by combining different components together in component groups. For instance in figure 3 the boxes E, F and G show a name group, address group and info group. This higher level of abstraction corresponds with part **c** in figure 2. Note that these component groups are UI representations for concerns in the problem domain (in this case the address book domain).

Switching to different devices, platforms, or even different user groups, might require a

```
inputComponent(?name,simple,?label,?l1,?l2) if
     label(?name,?label,?labelSpec,?l1),
     inputField(?name,?inputFieldSpec,?l2)

layout(?namelabellayout, 10, 15),
layout(?namefieldlayout, 80, 15, 250, 40),

inputComponent(comp1, simple, name, ?namelabellayout, ?namefieldlayout)
```

Fig. 4: InputComponent Rule

different implementation for certain entities. For instance, if an input component (D) of the type *date* no longer exists of a label and an input field, but has to be changed into a fancy calendar that pops up, the implementation of that input component has to be changed. Note that this is only possible if the two date formats are interchangeable because state and behavior of the interchanged components have to be consistent with each other.

2.2 UI and application interactivity

When separating UI and application, also the interactivity between the two needs to be separated (figure 1 number 3). As an example consider the address book with a birthday reminder included. It would be interesting to change the color of the birth-date field into yellow from one week before the birthday on. The fact that someone has a birthday within a week is part of the application (business) logic. Coloring the field yellow however is pure UI logic (figure 1 number 1), and thus the application has an influence on the UI behavior (figure 1 number 3). This application logic should not be hard-coded into the UI logic because this compromises reuse and thus it needs to be part of the interactivity layer (figure 2 number 3). Furthermore we might decide to change the behavior related to the birthday reminder, for instance instead of coloring the field yellow, an extra icon is added to the UI. Again the application logic hasn't changed, but the UI has and thus the interactivity between both is now different.

Another example is enabling and disabling buttons and fields, depending on the business logic. Remark that these examples might be simple and straightforward, but the interplay between a large set of these dependencies is nevertheless difficult.

2.3 UI Layout Strategies

Abstractions will also be used in order to adapt the UI layout. In figure 3 the UI is composed by adding the components together one below the other. However, another interesting layout might be to use another window for each component group, as shown by layout strategy 2 in figure 5. Or maybe one would like a tabbed window where each tab represents a component group, like layout strategy 3 in figure 5. This means that the unchanged component groups are combined by means of a different layout specifications. These different strategies are part of the layout strategy rule set in figure 2. Different layouting strategies are usefull in the context of multi-channel (e.g. for kids or grown-ups) and multi-platform (e.g.. PDA, internet, ...) adaptation of software user interfaces.

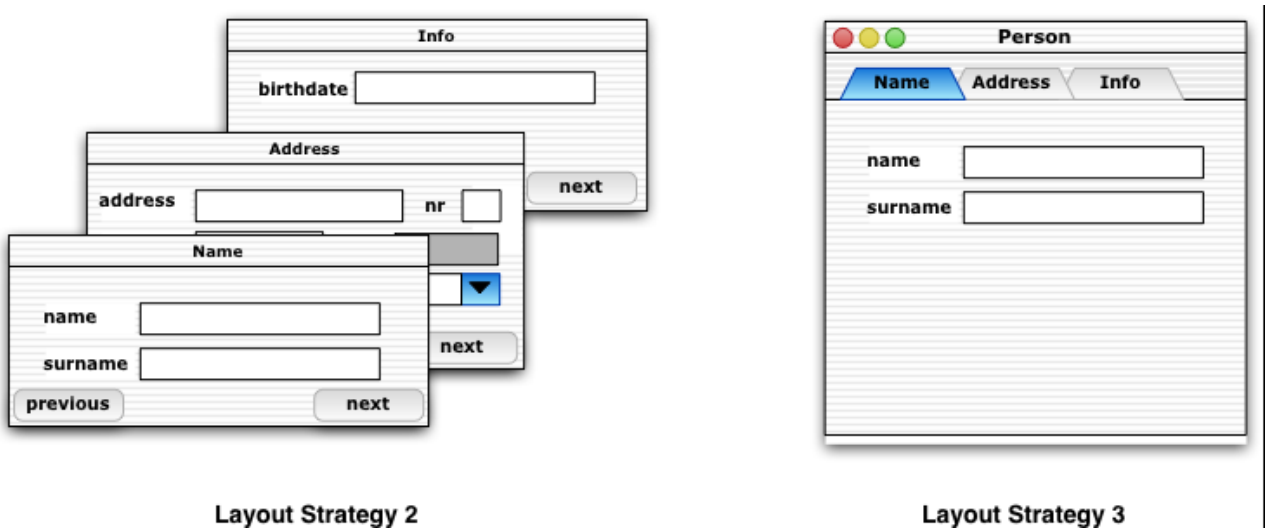

Fig. 5: Address book layouts

2.4 Reusing UI specifications

Composing User Interfaces by the use of abstract concepts (e.g. an address is composed of a street, zip code, city and country), which in their turn are composed with basic UI entities, allows better reuse of User Interface specifications. Both the abstract and specific concepts are reusable.

An abstract interface specification can be deployed towards different platforms by changing the specific implementation of the basic components for each of these platforms (figure 2 part **a**). The other UI building blocks remain unchanged. The responsibility for the platform specific representation is kept within the dedicated platform specific component. One UI for different platforms can thus be constructed based on the same higher level specification, but in combination with the different platform specific components an other UI will be the result. The same is true for reusing concrete concepts.

3 Possible Discussion topics for the workshop

Inheritance Reusing existing rule-bases can be done by parameterizing one rule-base by another one. This boils down to, for instance, replace component **a** in figure 2 by a new component **a'** such that part **b** and **c** still hold. In this case rule-base **b** will be parameterized by rule-base **a'**. We still need to investigate how to ensure inheritance between different UI sets, or how to make them dependent on each other. This is also related to evolution issues.

Customization and Configuration We want to generate the UI based on the declarative rules and facts. Nevertheless we should not restrict ourselves to static User Interfaces since a lot of the UI power is currently left unattended because of this staticness. It should be possible to customize a User Interface dynamically. For instance, when someone resizes a

window, another layout strategy might have to be applied. These layout strategies should be applied to the user interface dynamically, without need for recompiling the static UI.

Another issue related to this topic is putting constraints between the different UI building blocks [EL88, FB89]. For instance, how to specify that a certain component should be placed above another one (and what to do if conflicts arise)?

Evolving and Adapting It is not clear which abstractions will be needed. It is not possible to anticipate all possible abstract UI components, as this will depend on the developers point of view as well. Furthermore a certain abstraction might play a role in different UI framework parts, and it is not always clear where to draw the line. We acknowledge that it will never be possible to completely separate the UI from the application (since they have to interact at some point in time). But also on the level of the application interactivity layer it is not clear where to stop making abstractions.

Once the basic entities are defined, they are composed into more complex and abstract entities. A user will want to use these higher level entities without having to care about lower level entity descriptions. This implies that users will consider the high level entities as black boxes without knowing what the low level entities actually look like. Therefore two high level entities may contradict each other without the user knowing it.

Acknowledgements

We would like to thank Wolfgang De Meuter and Wim Lybaert for the interesting discussions and insights they provided us.

References

[EL88] E. Epstein and W.R. Lalonde. A smalltalk window system based on constraints. In *Proceedings of OOPSLA88*. ACMPress, 1988.

[FB89] B. Freeman-Benson. Constraint technologie for user-interface construction in thinglabii. In *Proceedings of OOPSLA89*. ACMPress, 1989.

[HL95] Walter Hürsch and Cristina Lopes. Separation of concerns. Technical report, Northeastern University, Boston, February 1995.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, June 1997.

[Szy98] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, January 1998.

[Wuy01] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Lab, Brussels, Belgium, 2001.