A case in Multiparadigm Programming : User Interfaces by means of Declarative Meta Programming

S. Goderis * W. De Meuter J. Brichau

Programming Technology Lab, Vrije Universiteit Brussel, Belgium

Abstract. Because there is currently no formal way to specify user interfaces, nor a clean way to decouple a user interface from its application code, we propose in this position paper the use of Declarative Meta Programming (DMP) to solve these problems. DMP uses facts and rules to write down a user interface in a declarative way, and will provide a more formal way to specify user interfaces. Furthermore DMP cleanly separates user interface from application code. The Declarative Meta Programming language SOUL that we intend to use, combines the declarative paradigm (for the user interface specification) and the object-oriented paradigm (for the application code). This position paper thus describes a case in multiparadigm programming.

1 Introduction

Although our knowledge concerning software engineering tasks has grown considerably during the last 20 years, few of these techniques are applied onto User Interfaces. Currently we distinguish two major problems with User Interfaces, namely

- the lack of a clean way to separate and couple the user interface and its underlying application, and
- the lack of a clear, powerful and uniform formalism to specify user interfaces.

Solving the first problem will benefit the independent evolution of the user interface and the application code. Even 'clean' separations like the Model-View-Controller pattern still have the problem that the underlying model code is interspersed with 'changed-messages'. A lot of user interface builders provide graphical tools for the development of user interfaces, but mainly they only generate stubs for procedures and methods. These stubs have to be coded manually by the software engineer in order for the user interface and the application to be coupled. Changing the user interface then results in manually changing this generated, and often unreadable, code. Current user interface builders lack a clean way to couple the user interface with the application code.

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

Solving the second problem will allow a non-programmer to build complete user-interfaces containing all necessary functionality that should otherwise be programmed. Furthermore, a uniform formalism for specifying user interfaces will benefit the porting of user interfaces to various graphical platforms (pc, mac, web, handhelds, mobiles,...). Since the desired formalism is to specifically express the *description* of user interfaces, a declarative formalism would be the most appropriate. A declarative *programming* language would even be better because it provides an executable declarative formalism and, as such, provides a user-interface specification language. The major advantage of using a complete language for specifying user interfaces is that it permits to write down more powerful descriptions of the dependencies and relations between the different user interface components. It also permits us to create abstractions from the low-level descriptions of user interface components to high-level descriptions of compositions of user interface components. Having different levels of abstractions will allow to easily replace one of the levels with a new set of declarations. Lower levels will be more platform dependent, and changing platforms will result in changing these lower levels while the higher levels can be kept. Thus there will be no need to rewrite the user interface specification.

In this position paper we state that creating a user interface for an application is a multiparadigm problem where

- the user interface is to be written down declaratively,
- the model is to be coded manually, and
- the coupling (how and where) of the user interface with the model is to be written down declaratively.

This approach requires the object-oriented paradigm for coding the model, and the declarative paradigm for writing down the user interface and the coupling. We will thus use a multiparadigm programming language combining both paradigms, namely a Declarative Meta Programming language. At Programming Technology lab several researchers have been using this paradigm for several purposes, such as code generation [2], co-evolution between design and implementation [8], aspect-oriented programming [1, 5], component-based development [7], etc. One of the artifacts build for this paradigm is the Smalltalk Open Unification Language (SOUL) which is the one we will focus on for our approach.

In section 2 we introduce the SOUL declarative programming language, which we will use in section 3 to specify and generate user interfaces. We conclude in section 4.

2 Smalltalk Open Unification Language

SOUL is a declarative meta layer on top of Smalltalk. It provides a prolog-alike programming language [4] and thus has all the properties of a declarative language. Logic facts are used to write down data or knowledge, while rules are used to reason about these facts and derive new facts. Furthermore SOUL

is implemented as a meta layer on top of Smalltalk, and it provides reflection and introspection operators [9]. Therefore it is possible to access the underlying Smalltalk system for retrieving information and adapting the underlying system according to the descriptions and rules at the upper level. This implies that, based on the rules and facts of the SOUL level, it is possible to generate code on the Smalltalk level. For more details (and the syntax) we refer to [8].

The Quoted term is a logic term in SOUL that we will most extensively use for our approach. It is a special logic term that was specifically included in SOUL to ease the manipulation of program source code by logic programs. It is similar to the quasi-quoted list in Scheme, which also represents programs as datastructures. The quoted term allows for writing down Smalltalk code as it is without it being evaluated. This construct can contain any kind of strings, possibly with some logic variables that are 'filled in' by the logic inference process. Templates (quoted terms with source code and logic variables) in combination with the substituted variables will result in 'real' code. Slightly changing the facts and rules will generate different code. In the following example firing the query `addClass` will result in a quoted term (between curly braces) containing the Smalltalk code that, if executed, will add a subclass `Matrix` to the super class `Array`.

```
addNewClass(Array, Matrix).

addClass({?super subclass: #?className }) if
   addNewClass(?className, ?super),
   class(?super)
```

3 DMP for User Interfaces

Model-based user interface development environments (MB-UIDEs) provide a context where developers can design and implement user interfaces in a systematic way, and more easily than when using traditional user interface development tools [6]. To achieve this aim, MB-UIDEs allow to describe the user interfaces through the use of declarative models. Pinheiro names three major advantages when using declarative user interface models [6] :

- A more abstract description of the user interface is provided ;
- User interfaces can be modelled using different levels of abstraction; the models can be refined incrementally; and user interface specifications can be re-used ;
- tasks related to the user interface design and implementation processes can be automated.

The idea behind MB-UIDE's is to be able to specify, generate and execute user interfaces.

As mentioned before we also want to use a declarative approach, namely Declarative Meta Programming, for specifying and generating user interfaces.

Since we tend to use SOUL, we intend to have a application coded in Smalltalk. SOUL itself will be used to write down the user interface specification and the coupling between user interface and application in a declarative way. We distinguish three parts of logic code, namely

- facts representing the user interface specification,
- facts representing the hooks in the application code, and
- rules expressing how to generate the code to combine both parts.

Hooks in the application code are certain points where calls to the user interface can be made, or where event handling (calls from the user interface) can be taken care of. Together with the coupling rules these hooks will ensure that the user interface is plugged onto the application. The user interface specification itself can consist of different layers, which allows for different abstraction levels.

3.1 User Interface specification

We will have to determine how the specification of different kinds of user interfaces can be put into a logic format, i.e. by the use of facts and rules. User interface specifications can range from elementary and simple knowledge (e.g. a title) to very advanced specifications (e.g. this circle always has to be centered in that rectangle). Elementary specifications are for example HTML and XML specifications, often used for web based user interfaces. Transforming these syntax-trees (what they basically are) into logic facts can easily be done by integrating a simple parser into SOUL. More advanced specifications that are represented by user interface builders can be transformed unambiguously into logic facts by means of transforming the structures used by these builders. Another kind of advanced specifications are constraints, which are, up to a certain degree, easily expressible by the use of logic programming.

For example a simple declaration for a user interface component `button` can be :

```
button(?name, ?label).
style(?name, ?style).
size(?name, ?size).
color(?name, ?color).
```

This specification indicates that there is a button with a name `?name` which has a label, a style (font of the label), size and a color.

A Formalism for User Interface specifications Declarative user interface specifications provide a more formal way of writing down user interfaces. This kind of formal specification is a major disadvantage of user interface builders these days. Graphical user interface builders provide a limited set of components which can not be altered by the user. Typically these builders use their own internal data structures, and the provided 'drawingtools' on top of these structures are not a

sufficient formal medium to specify user interfaces. A declarative approach combines the powerfulness of programming languages with the intuitivity of graphical tools. We do acknowledge that as an end tool a graphical user interface is wished for, but it can easily generate logic declarations.

For example, because SOUL is a programming language we can specify a rule for the previous example that declares all button properties to be obligatory :

```
isButton(?aButton) if
   button(?aButton, ?),
   style(?aButton, ?),
   size(?aButton, ?),
   color(?aButton, ?).
```

Different abstraction levels DMP also allows for different levels of abstraction. For the specification of user interfaces this is important because it will lead towards a more clean specification, but it also allows for easier generation of different kinds of user interface code. For instance we can specify a simple user interface consisting of a title, a textbox and two buttons :

```
title(uiTitle, 'This is a simple user interface').
textbox(goOn, 'Do you want to continue?').
button(yesButton, 'yes').
button(noButton, 'no').

color(uiTitle, green).
color(yesButton, grey).
color(noButton, grey).
size(yesButton, 2cm).
```

`color` and `size` specify properties of these lower level (possibly visual) components.

For often used combinations of components we can specify higher level concepts such as questions, actions, dependencies, listings, etc. For instance, the textbox and two buttons from the previous example express the higher level concept *question*. In logic this question can be specified in terms of a textbox and buttons :

```
question(q, 'do you want to continue?').

textbox(?name, ?questionText) if
   question(?q, ?questionText).
button(?name, yes) if
   question(?q, ?questionText).
button(?name, no) if
   question(?q, ?questionText).
```

The `textbox` and `button` rules are the transformation between the two layers of abstraction.

DMP also allows more powerful declarations. For instance the use of variables allows to express similar specifications by one specification only. For example, changing the color of the whole user interface to blue can easily be expressed by

```
color(? , blue).
```

instead of rewriting this fact for each element of the interface. This is extremely handy if we want to create a consistent user interface with the same 'look and feel' for multiple subcomponents.

We can also decide to let a property of a certain component depend on the property of another component. In

```
size(aTextbox, ?size) if
   size(aWindow, ?size).
```

the size of `aTextbox` depends on the size of `aWindow`. By the use of quoted terms it is possible in SOUL to let the size be a piece of Smalltalk code that will have to be executed at the time the size is really needed. For `aWindow` we could retrieve the size from the underlying Smalltalk system with

```
size(?aWindow, {?model  windowSize}) if
   model(?aWindow, ?model).
```

A more powerful example illustrating the same idea is the following :

```
relativeSize(?comp1, ?comp2, 0.5).

size(?comp1, {?model windowSize * ?ratio}) if
   relativeSize(?comp1, ?comp2, ?ratio),
   model(?comp2,?model).

size(?comp2, {?model windowSize}) if
   relativeSize(?, ?comp2, ?),
   model(?comp2,?model).
```

It expresses the dependency between two components where `comp1` has 0.5 times the size of `comp2`. Calculating the actual size of these components then depends on this relative size, and a size we supposedly retrieve from a class bound to `?model`.

Facts and rules referring to the underlying system are part of the coupling between the user interface specification and the underlying application code. These kind of rules and facts would have to reside at the lowest level of the user interface specification because it implies that it is known we are using Smalltalk user interface code. Using DMP for the generation of another kind of user interface code implies changing these levels. The lower level abstractions will be more user interface dependent (platform-dependent).

3.2 Generating User Interface code

As was stated by De Volder, DMP naturally supports Generative Programming [3]. After all we do want to be able to generate user interface code based on the user interface specification, whether it be HTML, Smalltalk code or something else.

In order to generate the user interface code, we will need to annotate the application code with logic facts which will represent the hooks onto which to plug the user interface. Using the code generation techniques of DMP, both user interface and application code will be linked. Since user interface configuration resides at the logic level, launching a query will generate the right code based on this configuration. Slightly changing the configuration will result in the generation of different code. The kind of user interface code that has to be generated will depend on the interface. For web based user interfaces it is possible that generating HTML or XML is sufficient, other interfaces will require a more advanced model (e.g. a Model-View-Controller pattern). These interfaces are independent from the underlying application and the different parts can be evolved and maintained rather independently from one another.

As an example, consider following rules that are part of a logic program that generates Smalltalk user interfaces from our specification in logic facts and rules:

```
method(?model, openView,
          { win := ?kindOfWindow new.
            win label: ?title.
            ?subwindowsAdditionCode win open}) if
   baseWindow(?uiName,?kindOfWindow),
   title(?uiName,?title)
   model(?uiName,?model),
   findall(?subwindowCode, subWindow(?uiName,?subwindowCode),
            ?subwindowsAdditionCode)


subWindow(?superId,
          {win addWindow: (?kindOfWindow new)
            atPositions: #(?x1,?x2,?y1,?y2). }) if
   embed(?superId,?subId,?x1,?x2,?y1,?y2)
   kindOfWindow(?subId,?kindOfWindow).
```

Launching the query `if method(?class,?methodName,?code)` will return all necessary methods that need to be generated in the Smalltalk application to build the user interface that we specified. A kind of code-generator will do exactly this and compile the resulting code. How the user interface window, with possible subwindows is to be created, is specified by using the quoted term. The kind of window that is to be generated, its title and model are specified by other logic facts. The same holds for the subwindow's position and kind of window.

We would like to stress that this is a tentative example and our approach needs to be elaborated. However, the same kind of technique was applied on components by De Volder [2].

4 Conclusion

Currently there is no way to easily specify uster interfaces formally, nor is there a clean way to decouple a user interface from its underlying application. In this position paper we propose the use of Declarative Meta Programming as a solution for these problems. DMP allows to write down user interface specifications in a declarative way by the use of facts and rules, and thus gives us a more formal way to write down user interfaces. In addition DMP cleanly separates the user interface from its underlying application and provides a means to generate the user interface code coupled with this application. Building user interfaces in this way is a case of multiparadigm programming. The declarative paradigm is combined with the object-oriented paradigm, and user interface specification is decoupled from the user interface code, and decoupled from the application code. We used SOUL as a multiparadigm programming language to put this into effect.

References

1. J. Brichau. Declarative meta programming for a language extensibility mechanism. In *ECOOP 2000, Workshop on Reflection and Meta Level Architectures*, 2000.
2. K. De Volder. *Type-Oriented Logic Meta Programming*. Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, September 1998.
3. K. De Volder. Generative logic meta programming. In *ECOOP 2001, Workshop on Generative Programming*, 2001.
4. P. Flach. *Simply Logical*. John Wiley and sons, 1994.
5. K. Gybels. *Aspect-Oriented Programming using a Logic Meta Programming Language to express cross-cutting through a dynamic joinpoint structure*. Bachelors thesis, Programming Technology Lab, Vrije Universiteit Brussel, August 2001.
6. P. Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In P. Palanque and F. Paternò, editors, *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, pages 207–226, Limerick, Ireland, June 2000. Springer-Verlag.
7. M. J. Presso. *Reflective and Metalevel Architecture in Java: from Object to Components*. Master thesis, Programming Technology Lab, Vrije Universiteit Brussel, 1999.
8. R. Wuyts. *A logic meta-programming approach to support the co-evolution of Object-Oriented design and implementation*. Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, January 2001.
9. R. Wuyts and S. Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *ECOOP 2001 International workshop on Multi-Paradigm Programming with Object-Oriented Languages*, 2001.