# Generating User Interfaces by means of Declarative Meta Programming

Sofie Goderis *  Wolfgang De Meuter
sgoderis@vub.ac.be  wdmeuter@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium

**Abstract**

A major problem still unsolved in computer science is how to specify a user interface whose code is cleanly separated from its underlying model in such a way that both the UI and the code can evolve independently from each other. Even 'clean' separations like MVC still have the problem that the underlying model code has to be 'polluted' by 'changed-messages'. In this position paper we propose Declarative Meta Programming (DMP) as a Generative Programming (GP) technique to configure user interfaces and their connection to the model in a declarative way. The code generation of DMP induces the UI code and its coupling to the underlying model. As such all the parts of the application can evolve independently from each other.

## 1 Introduction

In this position paper we present an approach for generating user interfaces in an automated way through the use of Declarative Meta Programming.

The last 20 years our knowledge about how to build software systems has grown by leaps and bounds. For instance in the object-oriented community some, at first new and revolutionary software engineering techniques became common sense, such as aspect oriented programming, design patterns, reuse diagrams, component-based development, etc. Furthermore computers and advanced software systems became part of our daily lifes, and now any person can be a potential user. People no longer have to be an expert in order to be able to use a computer. As a matter of fact, most users no longer have any technical knowledge nor experience. This lack of experience and knowledge has to be taken into account when developing software systems, which mainly means systems have to become more intuitive and straightforward in their use. Especially user interfaces play a major role towards achieving this goal for they provide the interaction between user and system. In addition, if the user interfaces of different applications would be somehow conform, the user will more rapidly learn how to work with a new system, and therefore accept it more easily.

Despite their potential and importance, the development of user interfaces has not been given enough attention. More precisely, writing down a user interface in a clear way has

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

not been explored, nor has it been investigated how to couple the user interface code with the application code. Although the Model-View-Controller pattern aims to decouple the user interface from its underlying application, the model is still interspersed with 'changed-messages' [Gyb01]. Other user interface builders that tend to help developing user interfaces (often graphically), generate the main skeleton of the user interface and stubs for procedures and methods. These stubs however have to be coded by the software engineer and as a consequence, changing the user interface results in manually changing the code. All these tools lack a distinct way of coupling the user interface with the application code.

User interfaces are build for a certain application, considering certain characteristics of the device on which the user interface is to be displayed. Until a few years ago an application was run on a typical device (e.g. computer, dishwasher, ...) and the user interface was developed for this one device only. Today on the contrary, it is possible to have one application run on different devices (e.g. computer, mobile phone, pda,...). For each of these devices the application needs a possibly different user interface. By writing down the user interface specification, and a device dependable specification we can generate the user interface, and it is no longer necessary to rewrite the user interface each time a new device has to run the application. In addition, this generative programming of user interfaces allows any user to specify an user interface without needing to know any technical characteristics of the application or devices.

Current user interface builders make use of rather low level concepts such as button, field, frame, .... Using higher level concepts, such as text, figure, listing, question, action, dependency, choice, ... raises creating a user interface to a higher level. This permits to use one (higher level) user interface for different devices. The actual transformation from higher level concepts to lower level concepts will be device dependable. For example, in a webbrowser a question might be printed on the screen, while on a handheld pda the question is spoken.

We suggest the use of Declarative Meta Programming, and De Volder [DV01] already argued that

> *Generative Programming is naturally supported by DMP by*
>
> - *using logic facts as DSL syntax to 'order' components.*
> - *using logic rules and constraints to characterize valid configurations.*
> - *using logic rules to map a client's request (problem space) onto a concrete build plan (solution space).*

More details about these arguments can be found in [DV01]. We want to investigate how to put this into practice, and more precisely for the generation of user interfaces.

In section 2 a short explanation of Declarative Meta Programming, and why it is useful for Generative Programming, is given. Section 3 will then give more details on how this paradigm will be used to carry our approach into effect. We conclude in section 4 .

## 2 Declarative Meta Programming

Declarative Meta Programming (DMP for short) aims to write down in a declarative way programs that deal with other programs. At the Programming Technology Lab, several re-

searchers have been experimenting with DMP and one of the artifacts that was developed for this purpose, is the Smalltalk Open Unification Language (SOUL) [Wuy98, Wuy01]. Although its original aim was to maintain a coherent evolution between the design and the implementation of a software system, it has been used, and proven useful, for other purposes as well. For instance, it was used for aspect-oriented programming [Bri00, Gyb01], component based development [Pre99], and code generation [DV98]. SOUL combines a declarative programming language with an object-oriented layer (i.e. Smalltalk).

## 2.1 Declarative Programming with SOUL

SOUL provides a prolog-alike programming language, and thus it 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. The following is an example of a small declarative program using SOUL. Note that apart from some notations SOUL is very similar to prolog [Fla94].

```
Fact parent(maria, eliza)
Fact parent(eliza, tom)

Rule grandparent(?agrandparent, ?agrandchild) if
     parent(?agrandparent, ?aparent),
     parent(?aparent, ?agrandchild)

Query grandparent(maria, ?someone)
```

By using the query `grandparent(maria, ?someone)` the program is run (the rule `grandparent(?x, ?y)` is triggered).

## 2.2 Meta Programming with SOUL

The interesting part about SOUL is that it is implemented as a meta layer on top of Smalltalk. Therefore it is possible to retrieve information from the underlying Smalltalk system, but moreover, the underlying system can be adapted 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.

**A Smalltalk term** is a logic term (just like variables, constants and functors) containing Smalltalk code that can be executed during logic interpretation. This allows us to wrap Smalltalk expressions and use them as logic constructs. There are two advantages :

- Smalltalk objects can be wrapped and used in SOUL as constants.
- Smalltalk expressions parametrized with logic variables can be wrapped and evaluated during the interpretation.

In the following example the smalltalk term `[Array]` is a Smalltalk term that wraps the Smalltalk class `Array` such that it can be used in SOUL. The query will test whether `Array` is a class or not.

```
Query class([Array])
```

When parametrizing the Smalltalk expression with logic variables, these variables will be substituted by their current binding during evaluation. If they are unbound, the interpretation of the smalltalk term fails. For example, the following

```
Rule selectors(?c, ?sels) if
     class(?c),
     equals(?sels, [?c selectors]).

Query simpleSelectors([ Array ], ?sels)
```

will result in a Smalltalk collection containing all method names of the class `Array`, which is the class bound to `?c`. `selectors` is a Smalltalk message understood by classes returning the methods of that class.

**The Quoted term** construct (between curly braces) allows to write 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 to be substituted for a string representation of their actual values before interpretation. Typically these quoted code terms are used to specify source code, and they allow for code generation. Using logic facts and rules acting on code templates (quoted terms with source code and logic variables), these templates combined with the substituted variables result in 'real' code. Note that the quoted term corresponds with a quasi-quoted list in Scheme, which as well is a way to represent programs as datastructures. In the following example firing the query `addClass` will result in 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)
```

The quoted term is the key element of SOUL that will allow Generative Programming in SOUL. It allows to write facts and rules whose querying results in code because binding the logic variables in the quoted term, as the reasoning process advances, results in 'real' code. By reconfiguring the logic layer (i.e. changing the rules), the logic query will generate different code.

# 3 Using DMP for generating User Interfaces

As we explained in the previous section, Declarative Meta Programming enables Generative Programming. Therefore we will use DMP as a means to configure and generate user interfaces in a generative programming way. When using DMP for the generation of User Interfaces we distinguish three important 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.

The user interface specification itself can consist of different layers. One layer will contain the higher level concepts of which the user interface is composed, another layer will transform these concepts into device dependable concepts. In DMP it is always possible to add extra abstraction layers if wished for. Therefore the device dependable layer can be reused for different user interface specifications, or the higher level specification can be put on top of the device dependable layers. 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. When generating the actual user interface code, the user interface specification is plugged onto these hooks.

### 3.1 User Interface specification

A first part of our approach is determining 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.

### 3.2 Generating User Interface code

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 on 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.

### 3.3 Example

Let us recall the idea of higher level concepts. The user interface is to specified by the use of higher level concepts such as figure, text, question, listing, action, etc. Another part of logic code will then generate the lower level specification depending on the chosen format (e.g. HTML, Smalltalk code,...). As a tentative example we consider the concept *question*. In SOUL we express this as follows :

```
Fact question(q, 'The actual question goes here').
```

The SOUL logic for generating a HTML specification where we simply print the question onto the screen, will then be :

```
Rule generateHTML(?something, ?code) if
     question(?something, ?text),
          equals(?code,
                { <html><body>
                  <h1>Question : </h1>
                  <ul> ?text  </ul>
                  </body></html>}
                  ).
```

Based on the same concept we can for instance also generate Smalltalk code that pops up a window showing the question:

```
Rule generateSmalltalk(?something, ?code) if
     question(?something, ?text),
     equals(?code,
                  { PopUpMenu confirm: ?text}).
```

We do acknowledge that this example is very simple, but it shows the aim and possibilities of our approach. Thanks to DMP we are quite confident though that this will work for complicated cases as well.

# 4 Conclusion

As user interfaces are gaining importance, new approaches are emerging such that the automatic generation of user interfaces can be improved. In this position paper we proposed the use of Declarative Meta Programming as an approach for writing user interface specifications down declaratively, and generating the user interface, together with plugging it onto the underlying application.

# References


[Bri00] Johan Brichau. Declarative meta programming for a language extensibility mechanism. In *ECOOP 2000, Workshop on Reflection and Meta Level Architectures*, 2000.

[DV98] Kris De Volder. *Type-Oriented Logic Meta Programming*. Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, September 1998.

[DV01] Kris De Volder. Generative logic meta programming. In *ECOOP 2001, Workshop on Generative Programming*, 2001.

[Fla94] Peter Flach. *Simply Logical*. John Wiley and sons, 1994.

[Gyb01] Kris 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.

[Pre99] Maria Jose Presso. *Reflective and Metalevel Architecture in Java: from Object to Components*. Master thesis, Programming Technology Lab, Vrije Universiteit Brussel, 1999.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings TOOLS USA98*, pages 112 – 124. IEEE Computer Society Press, 1998.

[Wuy01] Roel 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.