JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 9, Special Issue: TOOLS EUROPE 2007, October 2007

DEUCE : A Declarative Framework for Extricating User Interface Concerns

Sofie Goderis, Programming Technology Lab, Vrije Universiteit Brussel, Belgium
Dirk Deridder, System and Software Engineering Lab, Vrije Universiteit Brussel, Belgium
Ellen Van Paesschen, Laboratoire d'Informatique Fondamentale de Lille, France
Theo D'Hondt, Programming Technology Lab, Vrije Universiteit Brussel, Belgium

Evolving a software system not only affects the source code responsible for the core application, but also the user interface. Unfortunately user interface code is often scattered through and entangled with the application code. In large and complex user-interfaces, this tangling renders the implementation complex and hard to maintain. Currently, the application needs to perform both the necessary changes to the user-interface (e.g. disabling other buttons, propagating events, etc.) as well as invoke the required application logic. The Deuce framework (Declarative User Interface Concerns Extrication) intends to reduce the complexity of user-interface implementations by applying separation of concerns on three UI concerns : presentation logic, business and data logic, and connection logic. It does so by using a declarative meta-language (SOUL) on top of an object oriented language (Smalltalk) such that an adequate language is provided to describe the entire structure and behaviour of the user-interface, as well as to link it with the application.

1 INTRODUCTION

Current software systems need to exhibit a high degree of flexibility to accommodate a constant stream of change requests. This poses several challenges in standard business systems, but is taken to the extreme in the case of context-sensitive devices such as mobile phones and PDA's (a.k.a. Ambient Intelligence environments [9]). These devices put the software under continuous strain to adapt to different user capabilities, changing usage contexts, or even spatial information indicating whether it is being held in landscape or portrait mode. It is clear that this context-sensitivity does not only require special attention when writing the application's business and data logic. It also challenges the user interface (UI) implementation which needs to behave or present itself differently according to the current usage-context. As we will argue further on, the adaptation of the software is extremely complicated by the fact that the UI and the application code are closely entangled. Related work with respect to context-sensitivity [6] focusses on a language engineering approach,

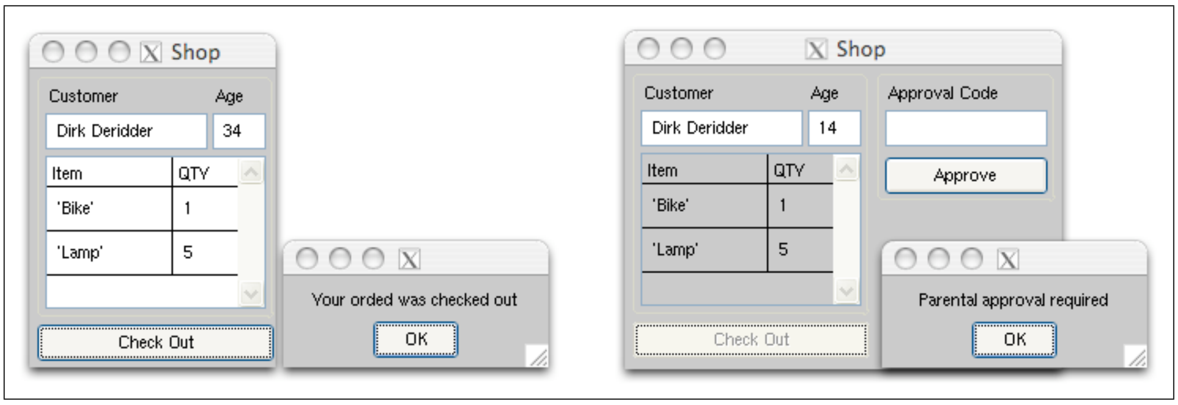

Figure 1: E-commerce requiring a Parent Approval for Minors upon ‘check out’

```
checkOut
    self age value < 18
        ifTrue:
            [self orderApproval
                ifTrue:
                    [self orderManager processOrder: self shoppingBag list.
                     Dialog warn: 'Your order was checked out'.]
                ifFalse:
                    [self disableCheckOutButton.
                     Dialog warn: 'Parental approval required'.
                     self enableApprovalForm]]
        ifFalse:
            [self orderManager processOrder: self shoppingBag list.
             Dialog warn: 'Your orded was checked out']
```

Figure 2: Smalltalk code for the ‘checkout’ procedure in the e-commerce example

but not on UIs.

Consider for example an e-commerce application where orders cannot be processed for children without the consent of their parents (see figure 1). Hence the application needs to provide a way to obtain parental approval if applicable. As a result of this requirement, the UI as well as the business logic are affected on several occasions. First of all adults should never be bothered with the existence of the approval procedure for their own purchases. Secondly, whenever a minor clicks on the checkout button, a (blocking) parental approval form must be shown. When a parent enters the correct approval code, a second click on the checkout button will start the standard checkout procedure. Handling the corresponding business and UI logic behaviour, is typically the responsibility of the event-handler connected to the checkout button. In figure 2 we illustrate the entanglement of several concerns in a naive Smalltalk implementation of the checkout procedure.

In blue (regular font face) we highlight UI related functionality such as the opening of a window and the enablement or disablement of particular buttons. In the future we refer to these types of concern as the *presentation logic concern.* Functionality that is not related to the UI, such as the business logic for verifying the age of a person, is shown in green (bold face). This concern is referred to as the *business and data logic concern.* The connection between the UI and the business logic is shown in red (italic). For example the code where the value of the UI component *shoppingBag* is provided as an argument for the *processOrder* procedure. This concern is named the *connection logic concern.* It is exactly the disentanglement of these different concerns which forms the subject of our work.

The entanglement and the associated evolution effort becomes worse if an existing implementation is adapted to incorporate new business requirements or to accommodate new contexts. Consider for instance an extension of the e-commerce application with a notion of discounts for returning customers. First of all, the fact that these customers can benefit from a discount should be highlighted on the order form (a UI visualisation concern). Secondly the code responsible for handling the checkout button needs to be intertwined with code which opens up an informative pop-up window (a UI behaviour concern). Moreover the same code should defer the calculation of the amount due to a different method (a connection logic concern).

In many cases programmers will apply advanced techniques in order to reduce the effect of code entanglement, such as the use of design patterns [12], dynamic object models [25] or state-machines [15]. Unfortunately this often results in the creation of an ad hoc infrastructure which introduces an additional maintenance hazard. Also it only delays and defers the problem of entanglement to a different time and place.

Existing UI approaches, including the ones based on the Model-View-Controller (MVC) architecture [23, 17], support a limited form of separation of concerns (see also section 7). More precisely such approaches focus on separating UI visualisation (in a view) from the business and data logic (in a model), and neglect the extrication of UI behaviour. As a result, evolving the UI's behaviour still requires browsing the source code, *manually* adding new UI behaviour and subsequently connecting it to the business logic where appropriate. In addition the code handling for the context-sensitive nature of the UI needs to be scattered throughout the application as well. This is why we advocate a built-in solution that covers the separation of all concerns in an integrated manner.

The *DEUCE* (Declarative User Interface Concerns Extrication) approach is based upon the well-known principle of *Separation of Concerns* (SoC) [16] and aims to provide programmer support for disentangling UI concerns. As already indicated, we focus on the following three concerns: presentation logic, business and data logic, and connection logic. In summary, inadequate support for separating the three UI concerns results in the following problems:

- *Evolving and maintaining the application is complicated.* Since all UI concerns are scattered throughout the business logic, the developer needs to browse through the code to make adaptations at different places in the code. This could easily break or even corrupt the existing functionality.

- *Reuse is difficult or even impossible.* Due to entanglement there exists an intrinsic connection between the presentation and the business logic. As a result it is not possible to reuse either the one or the other.

In the next section we introduce the underpinnings of the DEUCE approach which achieves the separation of concerns by means of a declarative language. Sections 3 to 6 present a proof-of-concept implementation for the important parts of DEUCE. Sections 7 and 8 discuss DEUCE in relation to related and future work.

2 DEUCE : DECLARATIVE USER INTERFACE EXTRICATION

As already introduced, DEUCE separates three main UI concerns which are illustrated in figure 3. Firstly, *UI presentation* covers concerns regarding the visualisation aspects and the behaviour aspects of the UI. Speaking in general terms, *UI visualisation* refers to how the UI looks and the widgets it contains (e.g. textboxes, buttons, labels). It also refers to the visual properties such as colour, enablement/disablement and state. For example, in the e-commerce application there is a “checkout” button of size 25 by 150 pixels with a label “check out”. DEUCE will actually contain a deification of the UI which includes the widgets and their properties. Visualisation also includes grouping components and specifying where components are positioned in relation to one another, i.e. layout. For instance positioning a component above another one, or putting all the components of one group in a single column. DEUCE will contain a description for these relations.

UI behaviour specifies what *actions* take place upon an *event* on a widget, as well as how widgets *influence* each other. For instance, when clicking on the checkout button, the order is processed and the user gets a notification.

Secondly, *business and data logic* with respect to the UI, specifies ‘hooks’ in the underlying code that link the UI with the application. These hooks describe where the application and its UI are connected such that one can be called from within the other. For instance, clicking the checkout button calls the checkout procedure.

Finally, *connection logic* makes the actual connection between the presentation and business and data logic. These connections can depend upon the context. For instance, if approval is needed, clicking the checkout button opens a parental approval form instead of continuing with the standard checkout procedure.

The programmer needs support for disentangling the several UI concerns. Therefore we need to satisfy the following requirements :

- **Requirement 1:** *A separate high-level specification for every concern.* Separating the concerns allows for changes to concerns in isolation. Because of the absence of disentanglement, concerns become possible reuse candidates.

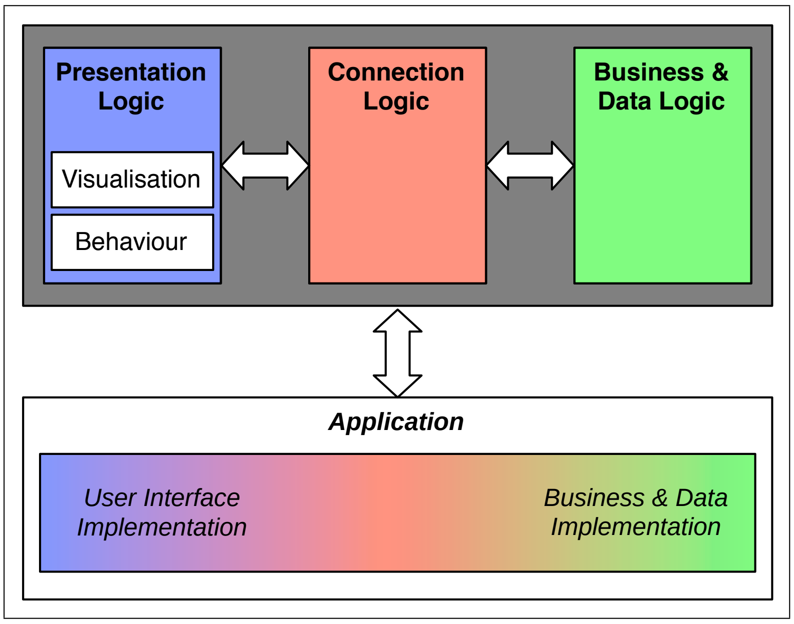

Figure 3: Separating UI Concerns with DEUCE

High-level specifications provide for a better understanding for the programmer since he now deals with 'domain concepts' of the UI instead of low-level technicalities of UI components.

- **Requirement 2:** *A uniform medium for expressing all the concerns involved.* This reduces the overhead for the programmer of having to learn several formalisms or mechanisms to specify the concerns.

- **Requirement 3:** *A mechanism to map the high-level entities onto the actual code level entities.*
The high-level specifications (requirement 1) have to be translated into the low-level UI specifics. Once a mapping between the two is established, the actual translation happens automatically. This allows for reusing the mapping, either because the UI has evolved or the mapping is reused amongst different high-level UIs that translate to a same low-level platform.

- **Requirement 4:** *An automated way to combine the different UI concerns.* The resulting application is created by combining the concerns with each other and the underlying business application. Providing an automatic mechanism for this combination is an important factor when offering support to the programmer.

To meet these requirements, we propose the *DEUCE* (Declarative User Interface Concerns Extrication) approach. It achieves a separation of concerns for UIs by using a declarative meta-programming (DMP) language for expressing and combining the concerns. Logic facts and rules describe and manipulate the UI concerns. The declarative reasoning mechanism of the declarative language puts the concerns and business logic together. As a proof-of-concept we implemented DEUCE. We use SOUL [28] as a declarative meta-programming language and the Cassowary constraint solver [2] for supporting automated layout (as a part of the presentation logic concern). Next, we shortly explain these declarative mechanisms.

Logic programming languages such as *Prolog* [10] typically involve *facts* for declaratively describing statements that are true and *rules* for indicating how the facts interact, and what implications may be taken from them. A *reasoning engine* is responsible for determining the set of applicable rules. A rule usually has the structure *IF condition THEN action or conclusion* or a variation thereof. The engine tries to match a rule's conclusion or condition with facts and other rules in order to come to a valid "solution". The benefit of logic programming is that the focus lies on what is to happen or to be described, and not how this is done.

Declarative Meta-Programming is an approach that provides a logic programming language as a meta-language on top of object-oriented programming languages. For instance a Prolog-like language on top of Java or Smalltalk. Several implementations have been developed of this approach, like *TyRuBa* [8] for Java and *SOUL* [28] for Smalltalk. Note that similar to Prolog, the order of the rules is important. SOUL makes it possible to retrieve information from the underlying Smalltalk system and Smalltalk objects can be wrapped in SOUL. Therefore, Smalltalk expressions can be used at the SOUL level and be parametrised with logic variables and evaluated during interpretation of the rules. Hence, SOUL is in *symbiosis* with Smalltalk. This symbiosis lies at the heart of the DEUCE implementation since it facilitates connecting high-level and low-level entities (see requirement 3).

The *Reasoning engine*, as part of the declarative programming language, makes use of a reasoning mechanism (algorithm). SOUL uses, in analogy to Prolog, a backward chaining algorithm. Other alternatives exist, such as forward chaining and regular expressions. For DEUCE we extended SOULs backward chainer with a constraint solver. However, what algorithm is used, is of no importance for DEUCE. What matters is that the reasoning mechanism uses the facts and rules of DEUCE's rule-base (i.e. the logical description of the UI) during its process such that a resulting application (i.e. the actual Smalltalk UI, linked with the application) is created. The reasoning mechanism is used at runtime to decide which UI behaviour to provide depending on the context (e.g. a parental approval procedure for minors).

The idea of a *constraint system to represent a UI layout* has been adopted in DEUCE as this has shown to be a very intuitive idea [19]. With DEUCE the UI is described (deified) at a higher logical level. This is also true for the layout relations between components, which makes adding/removing and repositioning components becomes less time-consuming. For instance, in figure 1 the approval form extends the original window, which can easily be done by making components visible/invisible. Nevertheless, if the obsolete components are positioned inbetween other components, making them invisible creates 'holes' in the interface. Removing the components at the high-level specification, will not necessarily change the abstract layout relations.

Given that these layout relations are translated into exact positions in the Smalltalk UI automatically, the 'holes' in the UI are avoided. SOUL's setup where a high level UI is 'downed' to the Smalltalk level through reasoning, created the opportunity to add automated layout to DEUCE. To do so a Smalltalk implementation of the Cassowary constraint solver [2] is used. A constraint is a logical relation between some variables with a possible restriction on the value domain of these unknowns. For instance, "a label should always be positioned to the left of an input field" is a constraint that connects two objects, label and input field, without specifying their exact coordinates. If the label or input field is repositioned, the constraint relation between the two should remain valid and thus the other object will also be repositioned. Constraints thus describe what relation should be maintained without specifying that relation computationally. "Calculating" a result by means of a constraint solver means searching for an actual solution with satisfied constraints.

In the following sections we zoom in on each of the aforementioned requirements and how they are fulfilled by our implementation of DEUCE.

3 A SEPARATE HIGH-LEVEL SPECIFICATION FOR EVERY CONCERN

Being able to abstract away from the low-level specifics of a UI, allows the programmer to better understand the UI and its flow. DEUCE facilitates the specification of the concerns at different layers of abstraction. A high-level layer can contain high-level (domain-specific) specifications for a certain application instance. For instance what components are part of the UI, how these relate to one another in both their visualisation and behaviour, and how they link to the underlying Smalltalk application.

DEUCE uses SOUL as a declarative medium to express the UI concerns. The underlying business and data logic is implemented in Smalltalk. SOUL's reasoning mechanism combines the several layers of declarative concerns with the Smalltalk code into the resulting application.

Presentation Logic Concern

The first concern, *presentation logic*, consists of visualisation and behaviour.

Visualisation deifies or 'ups' the actual UI such that its components and their visual properties (colour, label, . . .) can be accessed and updated from within SOUL. For example, figure 4 shows rules for describing the components in the e-commerce (standard) window and retrieving the contents of the `shoppingBag` component. The rules that automatically link the logic level with the Smalltalk level, are described in section 5.

As for the shoppingBag rule, the `component(shoppingBag, ?comp)` unifies `comp`

```
usedComponentsInInterface(<customerInfo, shoppingBag, checkOutButton>).


shoppingBag(?bag) if
   component(shoppingBag,?comp),
   contents(?comp, ?bag).
```

Figure 4: Presentation Logic : components and properties

```
group(customerInfo, <nameInput, ageInput>).
group(nameInput, <customerLabel, customerInputField>).
group(ageInput, <ageLabel, ageInputfield>)


above(customerInfo, shoppingBag).
above(shoppingBag, checkOutButton).

oneColumn(nameInput).
oneColumn(ageInput).
OneRow(<nameInput, ageInput>)
```

Figure 5: Presentation Logic : component relations

with the component that is linked with `shoppingBag` in the fact-base (actually being an upped UI fact). If this fact exists and unification succeeds, the `contents(?comp, ?bag)` will retrieve the contents of the shoppingBag component and unify it with the `?bag` variable. Hence, calling the query `shoppingBag(?bag)`, will unify `?bag` with the contents of the shoppingBag component.

Relations between components are also part of the visualisation, such as specifying that components belong together or specifying layout relations between components that position them left to one another, in one column. For instance in figure 5, grouping a label and an input field together to form a `nameInput`, or positioning the `shoppingBag` above the `checkOutButton`. These abstract layout relations are transformed by DEUCE into constraint relations. The Cassowary constraint solver resolves the constraints upon which DEUCE updates the layout property of the components.

Behaviour logic specifies how visualisation is affected by certain UI actions. For instance, clicking the `checkOutButton` requires a parental approval form to appear and some buttons to be disabled. In figure 6 the rule `activateApprovalRequest` specifies this UI behaviour, which will be linked with the clicking event through the connection logic concern (as explained below).

```
activateApprovalRequest if
   disable(checkOutButton),
   createDialog(warn, 'Parental approval needed'),
   enableApprovalForm.
```

Figure 6: Presentation Logic : UI behaviour

```
approvalRequired(?user) if
   child(?user),
   not(approved).

approved if
   [?inst orderApproval]
```

Figure 7: Business and Data Logic : hooks into the underlying application

Business and Data Logic Concern

A running application calls its UI and vice versa. ‘Hooks’ in the business application are places where this link with the UI takes place. These are described at the logic level by the *business and data logic* concern. This concern therefore represents part of the underlying business and data logic such that they can be used in the rules. For instance in figure 7, checking if a user received an approval to buy at the shop. The second rule is used to translate the high-level `approved` into a lower-level hook. `[?inst orderApproval]` is the actual hook into the application as it expresses a piece of Smalltalk code where the message `orderApproval` is send to the Smalltalk application (with `?inst` being bound to the current application instance). This mechanism is further elaborated on in section 5. Note that the `orderApproval` method is implemented as a standard Smalltalk method, and not as a logic rule.

Furthermore it is possible that the actual application hook depends on the context. The reasoning mechanism is used to determine what rule succeeds and therefore what hook to use. For instance in figure 8, an e-commerce application gives regular customers an extra discount. For these customers the application method calculating the discount will be different from the standard calculating method. The “calculate price” button for regular customers will therefore link to another calculating method, and use a different hook.

Connection Logic Concern

Connection logic is responsible for the actual connection between the presentation logic and business logic. For the actual mechanism creating the connection,

```
calculatePrice if
   regularClient(?user),
   [?inst calculatePriceWithDiscount].

calculatePrice if
   [?inst calculatePrice]
```

Figure 8: Business and Data Logic : hooks depending on context

```
model(checkoutButton, checkout).


checkout if
   approvalRequired(?user),
   activateApprovalRequest.


checkout if
   applicationProcessOrder,
   uiProcessOrder.
```

Figure 9: Connection Logic : linking checkout button with checkout query

we refer to section 5. In figure 9, the connection is called `model`, and links the `checkoutButton` with the `checkout` query. Clicking the button will launch this query and therefore the `checkout` rules as shown in figure 9 are triggered. If no approval is needed, either because the user is an adult or a child that already has approval, the first rule will fail and the second one will be triggered. Otherwise the first one will succeed. This means that, depending on the context (approval or not), different presentation logic and business logic actions are performed.

4 A UNIFORM MEDIUM FOR REPRESENTING ALL THE CONCERNS INVOLVED

As UIs are entangled with and scattered through the underlying business logic, evolving and maintaining the UI often results in the programmer spending a reasonable amount of time in browsing the code in order to get an understanding of where and how to make the necessary adaptations. A good separation of concerns eliminates this problem. The mechanism combining the several concerns into a solution, needs to consider all concerns at the same time in order to provide for an integrated solution. It therefore benefits from representing the concerns in a uniform medium. On top of this medium, dedicated tools are provided for the programmer, such that for each task an appropriate formalism can be used. For instance, in DEUCE the

```
componentName(?component,?compName) if
    isComponent(?component),
    equals(?compName,[?component name]).


disable(?compName) if
   componentName(?compName, ?comp),
   [?comp disable]
```

Figure 10: Presentation Logic : accessing and updating component properties

programmer can use the standard Smalltalk UI Builder to specify UI components, which DEUCE then translates into SOUL facts.

5 A MECHANISM TO MAP THE HIGH-LEVEL ENTITIES ONTO THE ACTUAL CODE LEVEL ENTITIES

The high-level UI specifications (requirement 1) are transformed into low-level and device/platform specific UIs. These transformation 'rules' can be reused by different high-level UIs that translate to UIs on the same platform. On the other hand, using different sets of transformation rules allow for reuse of the high-level UI and translate it to UIs for different platforms. Translating the high-level entities to the actual code level entities, or 'downing' the high-level UI specification to the Smalltalk level, happens through the use of SOUL's symbiosis mechanism. Smalltalk expressions, possibly annotated with logic variables, can be used at the SOUL level. (In the examples these expressions are between square brackets). The following examples show specifications at a lower level and make use of SOUL's symbiosis with Smalltalk.

Presentation logic deifies the low-level Smalltalk UI and uses symbiosis rules to access the components and their properties. For instance, to determine a component or its name property. Symbiosis is also used to change a component's properties, such as disabling a component (through the `model(checkoutButton, checkout)` fact in figure 10).

High-level UI components are 'downed' to Smalltalk UI components by 'standard' DEUCE rules. These rules are low-level, platform specific that are reused amongst different high-level UIs that all translate to Smalltalk UIs. Figure 11 illustrates a rule for adding an actionButton to the Smalltalk UI. The component's properties are queried and used in creating a Smalltalk specification for the button, which then is added to the `UIBuilder` of the Smalltalk UI.

Connection logic contains rules for linking presentation logic and business logic together, as shown in figure 9. The actual linking happens as shown in figure 12. The Smalltalk code behind a component, e.g. the checkout button, calls the SOUL

```
addSmalltalkComponent(?component, ?ui) if
    uiBuilder(?ui, ?builder),
    isActionButton(?component),
    componentName(?component,?name),
    text(?component,?text),
    layout(?component,?layout),
    model(?component,?model),
    equals(?smtComp,[ActionButtonSpec new name: ?name; setLabel: ?text;
                         layout: (?layout layout); model: ?model]),
    [?builder add: ?smtComp]
```

Figure 11: Presentation Logic : creating low-level UI components

```
applicationModel(?comp, ?code) if
   model(?comp, ?query),
   equals(?code,
      [(Soul.Evaluator eval: ('if', ?query asString)
           in: (Soul.Factory repository: #Deuce))nextResult])
```

Figure 12: Connection Logic : using a component's model to launch SOUL

evaluator to launch the query that was associated with the button in the first place (figure 9).

6 AN AUTOMATED WAY TO COMBINE THE SEVERAL UI CONCERNS

The declarative reasoning mechanism (that comes with a declarative programming language) uses facts and rules to come to a 'solution'. This process happens automatically. As this solution is the resulting application (with UI), the concerns are combined automatically into this application. Note that depending on the context, other rules will succeed and thus invoke other logic. This means that the UI flow path is 'calculated' automatically by the reasoning mechanism. Before this path was hard coded explicitly, for instance through 'if-statements'.

7 RELATED WORK

DEUCE brings several research areas together. First of all it aims for separation of concerns. To a certain extent, other approaches have also applied this principle to UIs. Automated layout in the presentation concern is crucial if the programmer wants to specify high-level interfaces and no longer needs to bother with low-level positioning of components. For this DEUCE uses the Cassowary constraint solver, but other possibilities exist. As for connection logic, we mention two approaches that solve problems of entanglement because of call-back procedures.

Separation of Concerns for UIs

The principle of separation of concerns has been applied, to a certain extent, to UI concerns by other approaches. However, these only focus on separating two of the concerns. The *Model-View-Controller (MVC)* architecture [24, 17] for example is a well-known approach, but is often misinterpreted such that MVC is thought of separating certain concerns but actually does not [11]. In MVC the controller handles input and transmits it to model and view. The view is the output towards the user, but it only covers the visualisation aspect of the UI. However, the behaviour concern, the business and data logic, and connection logic are captured by the model. These remain entangled, which gets even more stressed in Smalltalk's implementation of the MVC pattern [13]. For instance, when upon a button click other components get enabled, the enabling code is still entangled with the application code. When evolving UIs, this entanglement makes it difficult for the programmer to adapt the UI. *Model-View-Presenter* [21] is a generalisation of the MVC metaphor and is intended to overcome some of the problems with MVC as it is implemented in Smalltalk VisualWorks[5]. Unfortunately, in MVP they attribute the same meaning to model and view as in MVC. Business and data code and the UI behaviour concern are still entangled.

The *User Interface Markup Language (UIML)* is an XML-compliant language designated to build interfaces that can be deployed on multiple appliances [1]. Interfaces described with this declarative language consist of five parts : description, structure, data, style, events. These correspond to UI visualisation and connection logic. UIML separates the several UI concerns and provides rules to describe when to select what event. However these rules are fully 'matched' at specification time and cannot rely on a reasoning engine to reason with facts and other rules. Furthermore dynamic changes to the UI are not possible if not anticipated in advance. If several conditions are combined in order for an event to be triggered, they are combined statically and can result in long complex structures (similar to the if-statements for programmatic UI visualisation changes).

Model-based UI development environments divide a UI into four declarative models [7]. The application model describes the properties of the application that are relevant to the UI. The task-dialogue model describes what tasks a user can perform with the application as well as how these tasks relate to each other. The abstract presentation model provides a conceptual description of structure and behaviour of the visual parts of the UI. The concrete presentation model describes the visual parts of the UI in terms of widgets. Different model-based approaches (e.g. [27, 22, 14]) provide different techniques to specify (some of) these four models but not all of the approaches apply a same level of SoC. DEUCE can actually be considered to be a model-based approach where models are immediately executable.

Automated layout

Current research in automated layout [18] focusses on constraint-based and machine learning techniques, since both layout managers and templates are too limited. *Layout managers* [26] are part of UI toolkits and have simple layout policies built-in, such as horizontal or vertical layout. Designing complex hierarchical layouts are difficult and tedious. *Templates* [19], used in word-processing, focus on the format of the presented material (e.g. font, colour) and layout floating objects. Although most users of these systems overrule the simplistic placement policies for floating objects by placing the objects by hand. *Constraint-based automated layout systems* [4] deal with more advanced layouting possibilities and enforce position and size restrictions on components. A constraint solver is used to get to a solution, and thus a valid layout. *Machine learning techniques* [29] learn about what constraints to apply based on interaction with the user or by learning from a large provided set of presentations (i.e. layouts made by a layout-expert). DEUCE incorporates a constraint-based system for achieving automated layout.

Connection logic

Connecting the presentation concern with the business and data concern is achieved in DEUCE by its declarative mechanism. Other mechanisms exists but often do not provide, or provide only partially a mechanism to connect both concerns automatically.

Taps [3] are used to link the UI and the application and provide an extra level of indirection between the two. Taps partially map application objects to the interface objects and are triggered by user actions. The application and UI no longer know about each other, which makes it possible to easily change either one. However, we believe that the entanglement that before resided at application level, now resides in the tap.

Myers et al. [20] observe that a lot of call-back procedures perform no actual application work, but rather one of the following tasks : preparing data for the application, preparing data to be shown to the user, error checking and controlling connections between UI components. The authors present *Gilt*, a tool to generate expressions for these tasks automatically. Call-backs that do call application functions, are specified with high-level parameters instead of low-level widget properties. This is an extra indirection between the UI and application but only solves part of the connection concern.

8 CONCLUSION AND FUTURE WORK

Separation of Concerns is an important technique to come to possible more evolvable, reusable and maintainable code. Although User Interfaces would benefit of the same advantages, SoC is often limited or absent altogether. Three concerns related to UIs (presentation logic, business and data logic and connection logic) are discussed in this paper. To aid a programmer in achieving a separation of concerns we proposed DEUCE, an approach for Declarative User Interface Concerns Extrication. We have put the following requirements forward in order for providing programmer support for creating UIs and disentangling the concerns :

- **Requirement 1:** A separate high-level specification for every concern.
- **Requirement 2:** A uniform medium for representing all the concerns involved.
- **Requirement 3:** A mechanism to map the high-level entities onto the actual code level entities.
- **Requirement 4:** An automated way to combine the different UI concerns.

DEUCE uses declarative meta-programming for specifying the UI concerns and its reasoning mechanism to construct a valid UI out of this. A constraint solver is added to provide for automated layout. As a declarative meta-programming language DEUCE uses SOUL, a prolog-like language built on top of and in symbiosis with Smalltalk.

Requirement 1 is met because every concern is expressed by its own set of facts and rules. Within these rule-bases, different levels of abstraction occur, going from low-level platform specific towards high-level application specific. As we use the same declarative medium for expressing all the concerns, the programmer is provided with the uniform medium as proposed in requirement 2. The declarative reasoning mechanism that is part of SOUL, and hence of DEUCE, combines the facts and rules of the concerns rule-bases. Although different reasoning mechanisms (e.g. forward reasoning, regular expressions) can be considered as an alternative to SOUL's current backward chaining algorithm, they all combine the concerns automatically (requirement 4). Finally, requirement 3 is achieved through SOUL's symbiosis with Smalltalk. Smalltalk code can be called from within the SOUL level and used during the reasoning process. Through this mechanism high-level entities are transformed into actual low-level (Smalltalk) entities.

Because of SOUL's symbiosis with Smalltalk, we can switch from one to the other during runtime. Smalltalk code can be used at the SOUL level and SOUL queries can be launched from within the Smalltalk level. This allows for run-time interaction between the UI and the declarative reasoning mechanism. Contexts will be detected at runtime, and there is no need to anticipate all possible combinations of contexts. The reasoning mechanism can be asked to trigger all rules for which the conditions are met. DEUCE has only recently started to take advantage of these runtime possibilities, and further research and tool support in this area is required.

As the main focus of this research lies in separating UI concerns from a programmer's perspective, currently layouting is limited to a basic set of rules. In the future this set is to be extended with more advanced layouting schemes.

Also, even though the proposed solution is promising, further research needs to investigate the scalability on real world applications, as well as how rules and interaction schemes are to be reused.

REFERENCES

[1] M. Abrams, C. Phanouriou, and A. L. Batongbaca. Uiml : An appliance-independent xml user interface language. Technical report, Harmonia, Inc, 1999.

[2] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, 2001.

[3] T. Berlage. Using taps to separate the user interface from the application code. In *ACM Symposium on User Interface Software and Technology*, pages 191–198, November 1992.

[4] A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM Trans. Graph.*, 5(4):345–374, 1986.

[5] A. Bower and B. McGlashan. Twisting the triad: The evolution of the dolphin smalltalk mvp application framework. In *Tutorial Paper for ESUG 2000*, 2000.

[6] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *DLS '05: Proceedings of the 2005 conference on Dynamic languages symposium*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[7] P. P. 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.

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

[9] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J.-C. Burgelman. Scenarios for ambient intelligence in 2010. Technical report, EC Information Scociety Technologies Advisory Group (ISTAG), 2001.

[10] P. Flach. *Simply Logical*. John Wiley and sons, 1994.

[11] M. Fowler. Gui architectures, 2006.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : elements of reusabel object-oriented software*. Addison-Wesley, 1995.

[13] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[14] T. Griffiths, P. J. Barclay, N. W. Paton, J. McKirdy, J. B. Kennedy, P. D. Gray, R. Cooper, C. A. Goble, and P. P. da Silva. Teallach: a model-based user interface development environment for object databases. *Interacting with Computers*, 14(1):31–68, 2001.

[15] I. Horrocks. *Constructing the User Interface with Statecharts*. Addison Wesley Professional, 1999.

[16] W. Hürsch and C. Lopes. Separation of concerns. Technical report, Northeastern University, Boston, February 1995.

[17] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *JOOP*, August/September 1988.

[18] S. Lok and S. Feiner. A survey of automated layout techniques for information presentations. In *SmartGraphics Symposium*, pages 61–68, Mars 2001.

[19] S. Lok, S. Feiner, and G. Ngai. Evaluation of visual balance for automated layout. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*, pages 101–108, New York, NY, USA, 2004. ACM Press.

[20] B. A. Myers. Separating application code from toolkits : Eliminating the spaghetti of call-backs. In *UIST'91*, 1991.

[21] M. Potel. Mvp: Model-view-presenter - the taligent programming model for c++ and java. Technical report, Taligent Inc, 1996.

[22] A. R. Puerta and D. Maulsby. Mobi-d: A model-based development environment for user-centered design. In *CHI Extended Abstracts*, pages 4–5, 1997.

[23] T. Reenskaug. Models - views - controllers, december 1979.

[24] T. Reenskaug. Thing-model-view-editor : an example from a planningsystem. Technical report, 1979.

[25] D. Riehle, M. Tilman, and R. Johnson. *Dynamic Object Model.* Pattern Languages of Program Design 5. Addison-Wesley, 2005.

[26] SUN. Using layout managers. Java Tutorial, 2007.

[27] P. A. Szekely, P. Luo, and R. Neches. Beyond interface builders: Model-based interface tools. *Interchi*, April 1993.

[28] 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, January 2001.

[29] M. Zhou and S. Ma. Toward applying machine learning to design rule acquisition for automated graphics generation. Technical report, IBM Watson Research Center, 1999.

ABOUT THE AUTHORS

Sofie Goderis is a PhD student and research assistant at the Programming Technology Lab of the Vrije Universiteit Brussel, Belgium. She works on the use of Declarative Meta Programming for separating concerns in User Interfaces. See http://prog.vub.ac.be/~sgoderis/ for more information. She can be reached at sofie.goderis@vub.ac.be.

Dirk Deridder is a postdoctoral researcher at the System and Software Engineering Lab of the Vrije Universiteit Brussel. His research is supported by the Interuniversity Attraction Poles Programme - Belgian State - Belgian Science Policy. His PhD dissertation was entitled "A Concept-Centric Environment for Software Evolution in an Agile Context". See http://ssel.vub.ac.be/c3/ for more information. He can be reached at dirk.deridder@vub.ac.be.

Ellen Van Paesschen is an ERCIM fellow of the INRIA Futurs Jacquard Team at the Laboratoire d'Informatique Fondamentale de Lille in France. She obtained a PhD in Computer Science on July 10th 2006 at the Programming Technology Lab at the Vrije Universiteit Brussel. Her dissertation was entitled "Advanced Round-Trip Engineering: An Agile, Analysis-Driven Approach for Dynamic Languages". See http://prog.vub.ac.be/˜ellenvp/ for more information. She can be reached at Ellen.Vanpaesschen@lifl.fr.

Theo D´Hondt is a full-time faculty member of the computer-science department of the faculty of sciences of the Vrije Universiteit Brussel. He is responsible for the Programming Technology Lab, a software and language engineering research lab that goes back about 20 years. See http://prog.vub.ac.be/˜tjdhondt for more information. He can be reached at tjdhondt@vub.ac.be.