DEUCE : Separating Concerns in User Interfaces

Sofie Goderis
Programming Technology Lab
Vrije Universiteit Brussel
sofie.goderis@vub.ac.be

Dirk Deridder *
System and Software Engineering Lab
Vrije Universiteit Brussel
dirk.deridder@vub.ac.be

Ellen Van Paesschen
Laboratoire d'Informatique Fondamentale de Lille
University of Lille 1
Ellen.Vanpaesschen@lifl.fr

Abstract

As current software systems evolve continuously, both the application and its user interface (UI) have to be adapted. However, UI code is often scattered through and entangled with the application code. In large and complex UIs, this tangling renders the implementation complex and hard to maintain. The Deuce framework (Declarative User Interface Concerns Extrication) intends to reduce the complexity of UI 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 behavior of the UI, as well as linking it with the application.

1 Introduction

Current software systems have to show a continuous ability to adapt to new system requirements. This does not only affect the application's source code and business model, but also its user interface (UI). Evolving the application code as well as adapting the UI is complicated by the fact that the UI code is often entangled with the underlying application code. This makes creating and maintaining UIs a difficult task for the programmer. We propose to separate the UI and the application code as much as possible, resulting in a specific case of *Separation of Concerns* (SoC) [10]. More precisely, we focus on three different UI concerns: presentation logic (both UI visualization and UI behavior), business and data logic, and connection logic.

Some existing UI approaches, including the ones based on the Model-View-Controller (MVC) architecture [11] already support a limited form of SoC. Such approaches mainly focus on separating UI visualization (in a view) from the application logic (in a model), and neglect the extrication of UI behavior. As a result, evolving the UI's behavior often requires browsing the application source code, *manually* adding new UI behavior and subsequently connecting it to the application's code where appropriate. Additionally, the way UIs exhibit the appropriate visualization and behavior given a certain context, has to be hard-coded throughout the application.

We propose the *DEUCE* (Declarative User Interface Concerns Extrication) framework to apply the aforementioned SoC on object oriented systems. This is done by using a declarative UI language (SOUL) containing logic facts and rules which describe and manipulate the three UI concerns. The underlying application remains written in an object oriented programming language.

As a running example throughout the paper we will use the calculator application shown in Fig. 1. The standard version (fig. 1a) has buttons for number input, buttons for performing basic calculations and a log of previous calculations. A more minimalistic version (fig. 1b) works with whole numbers only. Therefore, divisions leading to a fractal result are prohibited by disabling all number buttons that result in a decimal if used as a second argument. A scientific version (fig. 1c) extends the standard version with extra operators.

In what follows we address separation of concerns for UIs (section 2). Section 3 explains DEUCE at a conceptual level, while section 4 explains a proof-of-concept implementation of DEUCE by means of scenarios for the calculator example. Section 5 addresses related work and a conclusion is given in section 6.

2 Separation of Concerns for UIs

We consider three concerns in UIs : presentation logic, business and data logic, and connection logic. *UI presentation* covers concerns regarding the visualization aspects and the behavior aspects of the UI. Speaking in general terms,

*supported by the IAP Programme - Belgian State - Belgian Science Policy

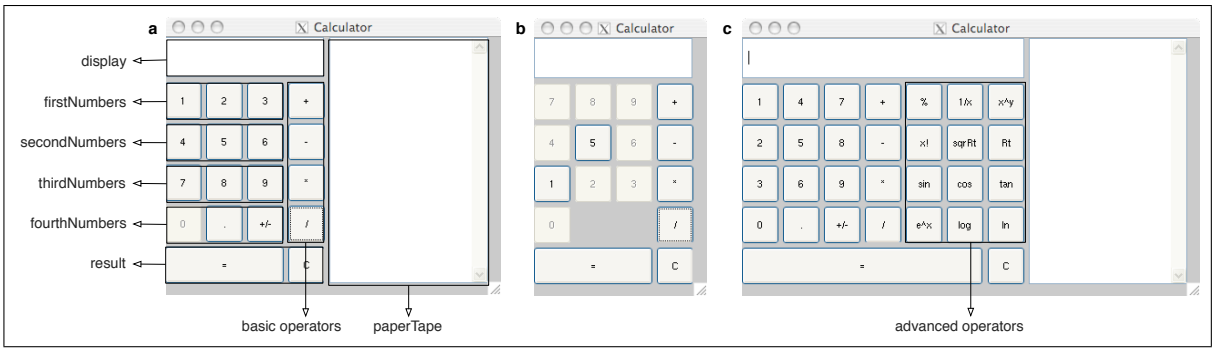

Figure 1. Three modes for a calculator : a) Standard b) Minimal c) Scientific

UI visualization 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 color, enablement/disablement, layout and state. *UI behavior* 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 equals button in the calculator (event), the result is calculated (action) and shown on the display (influence). 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 equals button calls an application method for actually calculating the result. Finally, *connection logic* makes the actual connection between the presentation logic and business and data logic. These connections can depend upon the context. For instance, in the minimal calculator the divide button does not allow divisions that result in decimal numbers. The combination of the three concerns, with the underlying business and data code, leads to the resulting application where both UI and application interact with one another.

Inadequate support for separating the three UI concerns results in the following main 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.

3 Declarative User Interface Extrication

The programmer needs support for disentangling the several UI concerns. Therefore we put forward 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 entanglement, specifications become possible reuse candidates. High-level specifications provide for a better understanding for the programmer since he now deals with the 'domain concepts' of the UI instead of the low-level technicalities of the UI components.

- **Requirement 2:** *A mechanism to map the high-level entities onto the actual code level entities.*
  The high-level specifications need 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 3:** *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 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 [19] for expressing and combining the concerns.

3.1 A separate high-level specification

Fig. 2 shows how UIs are created with DEUCE. The programmer implements the underlying business application as before (fig. 2, step 1) and specifies the declarative specifications for each of the concerns separately (fig. 2, step 2). These higher-level specifications abstract away from the low-level UI specifics and allow the programmer to better understand the UI and its flow. DEUCE facilitates the specification of the concerns at different layers of abstraction. At the highest-level, UI concern specifications are application-specific. For instance, for the standard calculator:

- Presentation logic: the calculator has number buttons, standard operator buttons, an input field, a log, etc. The log is positioned left-of the operator buttons, the operator buttons are put in one column, etc.
- Business and data logic: the application's divide method will be called by the UI.
- Connection logic: upon clicking the divide button, the divide method will be called.

A lower abstraction level expresses more-general rules, such as 'an input component consist of a label and an input field'. These rules are reusable amongst several UIs. The lowest level translates the high-level UI into a platform specific UI (see section 3.2). DEUCE combines the several abstraction layers with the underlying application (fig. 2, step 3) into the resulting application. This application uses DEUCE interactively at runtime to continue reasoning on the UI such that the UI responds to possible context changes (fig. 2, step 4).

3.2 From high-level to code-level entities

The high-level UI specifications 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. Typically these 'libraries' are reused by a high-level UI programmer but not implemented.

For instance, translations for the calculator into a Smalltalk calculator application include:

- Presentation logic: a number button is implemented with a Smalltalk *actionButton*.
- Business and data logic: the divide method is called by sending the Smalltalk message *divide* to the application behind the UI.
- Connection logic: Smalltalk code is plugged-in behind the event-handler of the divide button such that the right divide message is sent to the application, or such that the reasoning engine is re-launched.

The DEUCE framework provides rule-bases (fig. 2, step 5,) to deify (up) the actual UI to the declarative level, to reify (down) the declarative UI to the actual low-level UI, to achieve an automated layout and to connect the UI with the underlying application.

3.3 A uniform medium

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. Nevertheless the programmer should not have to get acquainted with several formalisms or mechanisms to specify the concerns. A uniform medium avoids this overhead. DEUCE uses the same declarative medium to express all of the concerns.

3.4 Automated way to combine concerns

The declarative reasoning mechanism uses facts and rules to come to a 'solution' (fig. 2, step 6). 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'.

4 Proof-of-concept Implementation

We currently have a proof-of-concept implementation of DEUCE. We use SOUL [18] as a declarative meta-programming language and the Cassowary constraint solver [2] for supporting automated layout (as a part of the presentation logic concern). We will shortly explain these declarative mechanisms. Next we illustrate DEUCE with some scenarios for the calculator example.

4.1 Declarative Mechanisms

Logic programming languages such as *Prolog* [7] typically involve *facts* for declaratively describing statements

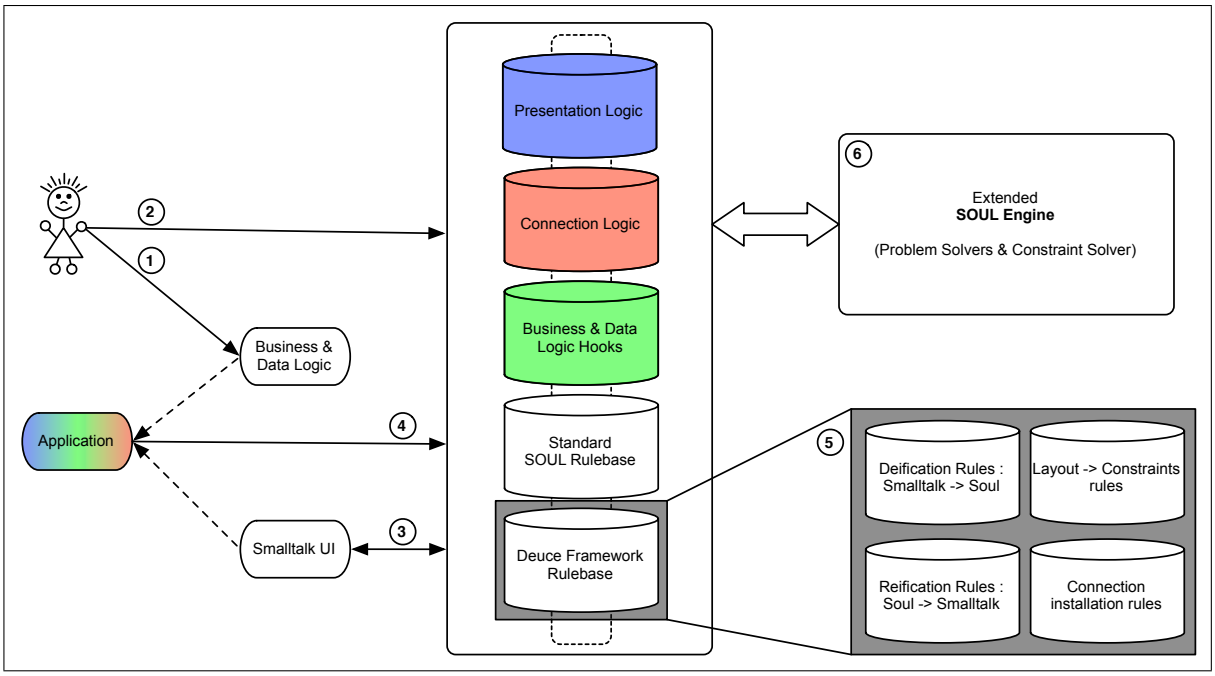

Figure 2. Creating UIs with DEUCE

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 on 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. 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.

The SOUL *Reasoning engine* makes use of a backward chaining algorithm. For DEUCE we extended SOULs backward chainer with a constraint solver.

Automated layout is achieved in DEUCE by using a *constraint system to represent the UI layout*. This idea has shown to be very intuitive [13]. With DEUCE the UI is described at a higher logical level. This is also true for the layout relations between components, which makes adding/removing and repositioning components less time-consuming. For instance, in Fig. 1 the scientific version extends the standard calculator. One could do this by making components visible/invisible. However, making the obsolete components invisible for the standard calculator, would create a gap between the operator buttons and the log. These 'holes' are undesired in a UI. Using the DEUCE constraint solver, these 'holes' are avoided.

4.2 Scenarios for a Calculator Application

The scenarios introduced next, illustrate functionality offered by the DEUCE framework. Each of these scenarios contain a number of steps or actions on the calculator. The user of the calculator will perform each of these actions, while the programmer is the one that provided the specifications of the UI and its behavior. For each step the corresponding support and functionality of the DEUCE framework is presented.

4.2.1 Scenario 1: Presentation logic visualization

Each calculator's presentation is expressed in DEUCE by specifying groups and components and layout relations. As illustrated in Fig. 1 for the standard calculator, several buttons are grouped into component groups. The numbers are

placed below the display and from top to bottom: firstNumbers, secondNumbers, thirdNumbers, fourthNumbers. To the left the operators are put in one column, and placed to the left of the log.

The scenario describes the functionality of DEUCE for supporting automated UI layouting. Consider the calculator example, initially in the standard mode.

- Step 1: The standard mode is changed to scientific mode via the UI. This requires changing the layout such that advanced operator buttons are added in-between the standard buttons and the log. *[DEUCE support]: With one rule it is specified which components are part of the scientific UI, as Fig. 3 shows.*

```
usedComponentsinInterface(<firstNumbers,
   secondNumbers,thirdNumbers,fourthNumbers,operators,
   advancedOperators,result,display,paperTape>)
```

Figure 3. Scientific: Components

- Step 2: The programmer specified that the number buttons are to be displayed top-down instead of left-right. The buttons in each number group are put in a column instead of a row. *[DEUCE support]: This is done through the advanced layout relation* oneColumn *in the scientific layout specification, as shown in Fig. 4.*

```
oneColumn(firstNumbers).
oneColumn(secondNumbers).
oneColumn(thirdNumbers).
oneRow(<firstNumbers,secondNumbers,thirdNumbers>)
```

Figure 4. Scientific: buttons top-down

4.2.2 Scenario 2: Presentation logic behavior

In this scenario we illustrate how the UI behavior is affected by context. Again consider the calculator, initially in the standard mode.

- Step 1: The calculator user pushes the '5' button. The application is called to store the first operand.
- Step 2: The user pushes the '/' button. The zero button gets disabled, since division by 0 is not allowed. *[DEUCE support]: The programmer specified that the corresponding UI logic to call for the calculator in standard mode, disables the zero button. See Fig. 5.*
- Step 3: The standard mode is changed to minimal mode via the UI. *[DEUCE support]: analogously to the previous scenario, the layout is automatically adapted for the minimal mode.*

```
divideButtonsDisabling if
   calculatorMode(standard),
   disable(zero)
```

Figure 5. Standard: '/' button behavior

- Step 4: The calculator user pushes the '5' button. The application is called to store the first operand.
- Step 5: The calculator user pushes the '/' button. As in the minimal mode of the calculator decimal numbers are excluded, division results that are decimal numbers are not shown to the user (Fig. 1, calculator a and b). 0 gets disabled together with all other numbers but 5 and 1. *[DEUCE support]: The programmer specified another rule expressing the UI changes for this context, namely when the calculator is in minimal mode. This is shown in Fig. 6.*

```
divideButtonsDisabling if
  calculatorMode(minmal),
  secondOperand(?x),
  disableButtonsForDivisionBy(?x, ?components),
  findall(?comp, disable(?comp), ?components)
```

Figure 6. Minimal: '/' button behavior

4.2.3 Scenario 3: Business and Data logic

In this scenario we illustrate applying different business behavior in different contexts, by showing a different business method to be called. Again consider the calculator, initially in the standard mode, depicted in Fig. 1.

- Step 1: The calculator user types '5' in the display. The application is called to store the first operand.
- Step 2: The user pushes the '/' button. The application awaits the second operand.
- Step 3: The user types '3' in the display. The application's divide method calculates the result. *[DEUCE support]: The programmer specified that the method to call when the divide button is clicked, is the 'divide' method. See Fig. 7.*
- Step 4: The standard mode is changed to minimal mode via the UI. *[DEUCE support]: analogously to the previous scenario the layout is automatically adapted for the minimal mode.*

```
behaviour(divideButton, #divide)
calculatorMode(standard)
```

Figure 7. Standard: Business&data link

- Step 5: The calculator user types '5' in the display field. The application is called to store the first operand.
- Step 6: The calculator user pushes the '/' button. The application awaits the second operand.
- Step 7: The user types '3' in the display field. The application's division method will now first check whether the second operand is valid before calculating the result. *[DEUCE support]: The programmer specified that the method to call when the divide button is clicked, is the 'divideMinimal' method. This is expressed declaratively as shown in Fig. 8.*

```
behaviour(divideButton, #divideMinimal).
calculatorMode(minimal).
```

Figure 8. Minimal: Business&data link

4.2.4 Scenario 4: Connection logic

In this scenario we illustrate how business logic and application logic are put together. Again consider the calculator. The following scenario applies for all calculator modes.

- Step 1: The calculator user pushes the '5' button. The application is called to store the first operand.
- Step 2: The user pushes the '/' button. Both scenario 2 and 3 apply. *[DEUCE support]: The programmer specified (scenario 2 and 3) that the UI has to disable buttons and what method has to be called in the application. The declarative reasoning mechanism decides which rules apply. See Fig. 9.*

```
divide if
   divideButtonsDisabling,
   behaviour(divideButton, ?x)
```

Figure 9. Presentation and Business logic

5 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. As for connection logic, we mention two approaches that solve entanglement problems related to callback 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. The *Model-View-Controller (MVC)* architecture [16, 11] for example is a well-known approach, but is often misinterpreted such that MVC is thought of separating certain concerns but actually does not [8]. In MVC the controller handles input and transmits it to model and view. The view covers the visualization aspect of the UI. However, the behavior 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 [9]. *Model-View-Presenter* [15] is a generalization of the MVC metaphor and is intended to overcome some of the problems with MVC [5]. Unfortunately, MVP attributes the same meaning to model and view as in MVC.

The *User Interface Markup Language (UIML)* is an XML-compliant language designated to build interfaces that can be deployed on multiple appliances [1]. 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.

Model-based UI development environments divide a UI into four declarative models [6]. 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 behavior 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 provide different techniques to specify (some of) these four models but not all of the approaches apply a same level of SoC. DEUCE can be considered to be a model-based approach where models are immediately executable.

Automated layout Current research in automated layout [12] focusses on constraint-based and machine learning techniques, since both layout managers and templates are too limited. With *layout managers* [17] designing complex hierarchical layouts are difficult and tedious. *Templates* [13] use simplistic placement policies that are overruled by most users 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* [20] 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).

Connection logic *Taps* [3] are used to link the UI and the application and provide an extra level of indirection between the two. However, we believe that the entanglement that before resided at application level, now resides in the tap.

Myers et al. [14] 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.

6 Conclusion

Separation of Concerns is said to lead to evolvable, reusable and maintainable code. Although User Interfaces would benefit from the same advantages, SoC is often limited or absent altogether. Three concerns related to UIs (presentation, business and data, and connection logic) are to be separated from one another. To aid a programmer in achieving a full SoC we propose a framework called DEUCE. This framework uses logic meta-programming for specifying the UI concerns and uses its reasoning mechanism to construct a valid UI out of this. A constraint solver is used to provide for automated layout, since otherwise reuse becomes unfeasible for the programmer.

We believe that DEUCE will aid in a full separation of concerns of the UI concerns, such that the task of creating, but especially maintaining and evolving, UIS becomes less complex for the programmer.

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*, 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.

[6] 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*, 2000. Springer-Verlag.

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

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

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

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

[11] 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.

[12] S. Lok and S. Feiner. A survey of automated layout techniques for information presentations. In *SmartGraphics Symposium*, Mars 2001.

[13] 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*, 2004. ACM Press.

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

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

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

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

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

[19] R. Wuyts and S. Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *ECOOP 2001 International workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.

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