Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN
Vakgroep Informatica
Laboratorium voor Programmeerkunde

On the Separation of User Interface Concerns

A Programmer's Perspective on the Modularisation of User Interface Code

Proefschrift voorgelegd voor het behalen van de graad van Doctor in de Wetenschappen

Sofie Goderis

Academiejaar 2007 - 2008

Promotoren: Prof. Dr. Theo D'Hondt en Dr. Dirk Deridder

VRIJE UNIVERSITEIT BRUSSEL • SCIENTIA VINCERE TENEBRAS •

Print: DCL Print & Sign, Zelzate

VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. ++32 (0)2 289 26 50
Fax ++32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5487 497 3
NUR 980
Legal deposit D/2008/11.161/041

Abstract

The subject of this dissertation is the modularisation of user interface code. We provide a conceptual solution which achieves the separation of the three user interface concerns we distinguish, namely the Presentation, Application and Connection concerns. As a proof-of-concept we have implemented DEUCE which uses a declarative programming language to specify the various user interface concerns. The declarative reasoning mechanism is used in combination with meta-programming techniques in order to compose the various user interface concerns and the application into a final runtime application.

An important force within the domain of software engineering is the principle of separation of concerns. It is applied to modularise software systems such that various parts of the system can be dealt with in isolation of other parts. As modularisation decreases the amount of code tangling and code scattering, it increases the maintainability, evolvability and reusability of the various parts. Also within the domain of user interfaces this principle of separation of concerns can be applied for modularising user interface code and application code. However, the principle of separation of concerns has not been applied to a full extent to user interfaces yet. The developers are responsible for implementing both the application part and the user interface part, and the link between both parts. Furthermore, with the rise of new software challenges such as agile development environments and context-sensitive systems, these software systems and their implementation need to exhibit an increasing degree of variability and flexibility. Developing them still requires the developers to deal with tangled and scattered user interface code. Hence, evolving and maintaining user interface code and its underling application has become a cumbersome task for the developers. This task can be alleviated by supporting the developers to apply the principle of separation of concerns to user interface code in order to improve the evolvability and reusability of user interface and application code.

While there exist contemporary techniques that are geared towards solving the problem of modularising user interface code and application code, these typically fall short in several ways. For example, the model-view-controller pattern which is still a major player when it comes to separating user interface code from application code, does not tackle the problem of code tangling and scattering to a full extent. Other approaches such as Model-Based User Interface Development and User Interface Description Languages focus on a generative approach and are typically used for software systems where all interactions with the application originate from within the user interface only.

The solution we propose deals with separating the user interface code from its underlying application code for software systems in which application and user interface interact in both ways, for which several views can exist at the same time and in which dynamic user interface changes or updates are necessary. We elaborate on a conceptual solution to separate the following three user interface concerns. The presentation concern is related to the user interface itself and represents what the interface looks like and how its behaves. The application concern specifies how the application is triggered from within the user interface, and vice versa. The connection concern expresses how the presentation concern and application concern interact with each other and creates the link between both parts. We also postulate five requirements that are crucial for any solution that is aimed at a systematic modularisation of user interface code.

As a proof-of-concept implementation, we provide DEUCE (DEclarative User interface Concerns Extrication). This proof-of-concept uses a declarative meta-language (SOUL) on top of an object-oriented language (Smalltalk) and by doing so it provides a specification language to describe the entire structure and behaviour of the user-interface as well as its link with the application. This specification of the user interface concerns and the underlying application code are automatically composed into an final application for which a dynamic link with the original UI specification is maintained. DEUCE is put to practice by refactoring a Smalltalk personal finance application in order to validate that the conceptual solution does achieve an improved modularisation of user interface code. This modularisation removes code scattering and makes code tangling explicit in one location. Hence, the conceptual solution proposed in this dissertation establishes the separation of user interface concerns from a programmer's perspective.

Samenvatting

Deze doctoraatsverhandeling handelt over het modulariseren van user interface code. We bieden een conceptuele oplossing aan die er toe bij draagt om de drie user interface bekommernissen presentatie, applicatie en connectie bekommernis, van elkaar te scheiden. Om deze conceptuele oplossing te staven, implementeren we het prototype DEUCE. Dit prototype maakt gebruik van een declaratieve programmeertaal om de verscheidene user interface bekommernissen afzonderlijk van elkaar te beschrijven. Het redeneermechanisme achter deze declaratieve taal wordt gecombineerd met meta-programmeer technieken ten einde deze bekommernissen samen met de onderliggende applicatie te combineren tot het uiteindelijke software systeem.

Het principe van het scheiden van bekommernissen speelt reeds een belangrijke rol binnen het domein van de software engineering. Het wordt toegepast om software systemen te modulariseren zodat de verschillende onderdelen van dergelijk systeem afzonderlijk van elkaar beschouwd kunnen worden. Hierdoor zal de code die betrekking heeft op één bekommernis niet langer verspreid staan over de gehele implementatie en zullen bekommernissen niet langer sterk met elkaar verweven zijn. De modularisatie van bekommernissen verhoogt de herbruikbaarheid en de evolueerbaarheid van het software systeem. Het principe van het scheiden van bekommernissen kan ook op user interfaces toegepast worden zodat user interface code en applicatie code van elkaar gescheiden worden. Echter, tot op heden werd dit niet ten volle uitgebuit en software ontwikkelaars zijn nog steeds verantwoordelijk om beide delen te implementeren alsook er voor te zorgen dat beide delen samenwerken.

Het toenemende belang van nieuwe software uitdagingen, zoals agile development en context-sensitieve systemen, versterken de noodzaak om een goede scheiding van user interface code te bewerkstelligen. Immers, de implementatie van deze software systemen moet een steeds groeiende flexibiliteit aan de dag leggen. Echter, tijdens het ontwikkelen van deze systemen worden software ontwikkelaars nog steeds geconfronteerd met verspreidde en verweven user interface code. Bijgevolg blijft ook hier het ontwikkelen, evolueren, onderhouden en hergebruiken van user interface code en de bijhorende applicatie code een moeilijke taak. Het principe van scheiden van bekommernissen ten volle toepassen, biedt ondersteuning voor de software ontwikkelaars bij het creëren van dergelijke systemen.

De implementatie technieken die vandaag de dag worden toegepast om user interface code te modulariseren, falen helaas op meerdere vlakken. Bijvoorbeeld het model-view-controller patroon wordt op heden nog veelvuldig gebruikt om user interface code van applicatie code te scheiden. Desalniettemin wordt hiermee het probleem van de software ontwikkelaars niet opgelost aangezien zij op implementatie niveau nog steeds geconfronteerd worden met het verspreid en verweven zijn van user interface code. Andere aanpakken, zoals Model-Based User Interface Development en User Interface Description Languages gebruiken een generatieve aanpak om vanuit een user interface beschrijving een applicatie te genereren. De dynamische flexibiliteit alsook de link vanuit de applicatie naar de user interface gaat bij deze benaderingen vaak verloren.

In deze doctoraatsverhandeling wordt een oplossing aangeboden om de scheiding van user interface code te bewerkstelligen voor software systemen waarbij de user interface en de applicatie in twee richtingen kunnen inter-ageren, waar meerdere user interfaces voor eenzelfde applicatie op hetzelfde ogenblik in gebruik kunnen zijn, en waarbij dynamische user interface aanpassingen gewenst zijn. We lichten de vijf vereisten toe waaraan voldaan moet worden om een gedegen scheiding van user interface code voor dergelijke systemen te bekomen. De user interface bekommernissen die we hierbij onderkennen zijn de presentatie, applicatie en connectie bekommernis. De presentatie bekommernis geeft weer hoe de user interface er uit ziet en zich gedraagt. Hoe de applicatie wordt aangeroepen vanuit de interface, en vice versa, wordt uitgedrukt door middel van de applicatie bekommernis. De connectie bekommernis tenslotte geeft aan hoe beide voorgaande bekommernissen samen gebracht worden.

De conceptuele bijdrage van deze thesis wordt in de praktijk gebracht door middel van een prototype implementatie, DEUCE genaamd. DEUCE staat voor "DEclarative User interface Concerns Extrication" en maakt gebruik van een declaratieve metaprogrammeer taal (SOUL) bovenop een object georiënteerde taal (Smalltalk). Op deze manier voorziet DEUCE in een specificatie taal om de structuur en het gedrag van een user interface te beschrijven, alsook om de link tussen de user interface code en de applicatie code uit te drukken. DEUCE voorziet tevens de mechanismen om deze specificaties te combineren met de onderliggende applicatie om tot het uiteindelijke software systeem te komen. Om onze aanpak te valideren, gebruiken we DEUCE om de persoonlijke financiën applicatie *MijnGeld* deels te her-implementeren. Hiermee tonen we aan dat de conceptuele oplossing in de praktijk gezet kan worden onder de vorm van DEUCE en inderdaad de benodigde modularisatie van user interface code kan bewerkstelligen. Deze modularisatie elimineert de code verspreiding en concentreert de code verwevenheid expliciet op één plaats. Bijgevolg zorgt de conceptuele oplossing zoals voorgesteld in deze doctoraatsverhandeling voor een betere scheiding van user interface bekommernissen vanuit het perspectief van de software ontwikkelaars.

Acknowledgements

The acknowledgements is probably the most read section of a PhD thesis. After all one can only get through the process of a PhD thanks to the support of many others. Also this dissertation would not have been what it is without the tremendous support that I have received from my colleagues, friends and family.

I thank Theo D'Hondt for being the promoter of this thesis during all those years. He gave me the opportunity to become a researcher at PROG and provided me with the means to pursue this work. Despite all the chores that come with the job of being a professor and a dean, Theo always finds a way to keep track of his people and to stimulate us to grow, both as a person and as a researcher.

I am also deeply indebted to my promoter Dirk Deridder. He became involved in this project whilst still finishing his own PhD. Even though it was not always straightforward to find the time to work together, Dirk managed to support and motivate me throughout these last years. Not only did he read this thesis text over and over again, he never gave up on me during hefty discussions. Knowing me, this does take a lot of courage.

A special thanks goes to Wolfgang De Meuter. During my last year of my master in computerscience he convinced me to do my thesis under his guidance. Without that thesis, no research microbe would have bitten me and I would not have started on the path towards a PhD. Furthermore he helped me to take the first steps on this path as he got me through writing and defending an IWT scholarship proposal. Finally, Wolf is also the inspiration behind EMOOSE. This master program gave me the international experience I longed for after my studies and it taught me to value the computer-science curriculum at the VUB as well as its environment.

I thank Robert Hirschfeld, Karin Coninx, Viviane Jonckers, Olga De Troyer, and Wolfgang De Meuter for finding the time to be on my thesis committee and for their insightful comments and suggestions.

Thanks also to Andy Kellens and Kris De Schutter for proof-reading this dissertation, up to several times. I know I am not easy to convince, but I do appreciate the effort as it improved the quality of the text drastically.

A thank you is also more than appropriate to all the colleagues I have met at PROG. At every stage of the PhD process their research perspectives broadened mine. Furthermore this last year my colleagues provided me with the opportunity to focus entirely on writing this dissertation. Thank you Adriaan Peeters, Andoni Lombide Carreton, Andy Kellens, Bram Adams, Brecht Desmet, Charlotte Herzeel, Christian Devalez, Christophe Scholliers, Coen De Roover, Dirk van Deun, Elisa Gonzalez Boix, Ellen Van Paesschen, Isabel Michiels, Jessie Dedecker, Johan Brichau, Johan Fabry, Jorge Vallejos, Kim Mens, Kris De Schutter, Kris De Volder, Kris Gybels, Linda Dasseville, Matthias Stevens, Pascal Costanza, Peter Ebraert, Roel Wuyts, Stijn Mostinckx, Stijn Timbermont, Thomas Cleenewerck, Tom Mens, Tom Tourwe, Tom Van Cutsem, Werner Van Belle, Wim Lybaert, Yves Vandriessche, Lydie Seghers, Brigitte Beyens, and Simonne De Schrijver.

Almost last but not least, my biggest gratitude goes to my parents. Ma and Pa, you told me not to mention your names in this acknowledgements, but one of the advantages of writing a PhD is that one gets to explicitly thank people that otherwise are taken for granted. You have always stimulated me to find my own way by giving me the opportunity to pursue my own dreams and ideas. At every step of the road you are there with advise and everlasting encouragement. It is because of you that I could accomplish this PhD and that I became the person that I am today.

Finally, in the course of years I have met you, my friends. Some of you have been around for a long time and some of you I have only recently met. Nevertheless all of you provided me with the necessary distractions when I needed an outlet. It is because of you that I kept finding the energy to go on in difficult times.

Thank you!

Sofie Goderis
3 juli 2008

Table of Contents

Chapter 1
Introduction

In current software systems, variability has come to play an important role. Software variability is defined as *the ability of a software system or artefact to be efficiently extended, changed, customised or configured for use in a particular context* [Sva05]. For example, software varies because it adapts to changing requirements in various contexts. This is taken to the extreme in context-sensitive devices such as mobile phones and PDA's (a.k.a. Ambient Intelligence environments [Duc01]). 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.

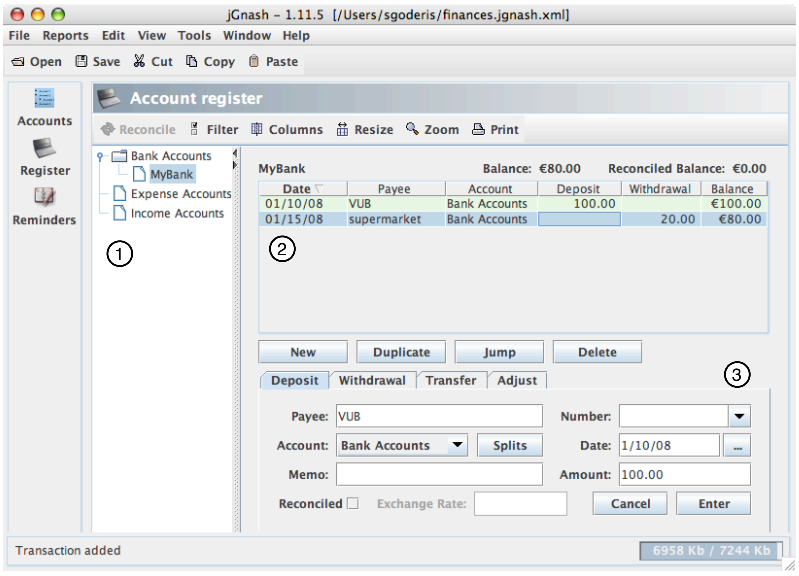
While a high-degree of variability will allow software to be reused in a broader range of contexts, it implies that the implementation needs to exhibit a high degree of flexibility. This does not only require special attention when writing the application's logic but it also challenges the user interface implementation which needs to behave or present itself differently according to the current usage-context. The increase in diversity of both the types of computing devices in use as the task contexts in which they operate, implies a major change in user interfaces [Mye00]. Context of use refers to the user stereotype, any computing platform, and the physical environment in which the user is carrying out a task with the designated platform [Mic08]. For instance for a novice user an application can provide a user interface with a step by step wizard interface that gives the user more guidance. When an application switches from a desktop computer to a mobile PDA, the computing environment changes and has an impact on the display of the application. Changing the physical environment includes switching from an office to a home situation, registering changes in lighting and temperature, switching from landscape to portrait mode, and many more. Such changes can also have their impact on the user interface, in addition to changes on the application's behaviour. With the rise of context-sensitive user interfaces, also the exploitation of user interface design knowledge at run-time becomes important as these user interfaces adapt to the context and need to reflect dynamic changes in context into the user interface when appropriate [Dem06, VdB05].

Developing applications and their user interfaces such that they support the necessary flexibility is extremely complicated by the fact that the user interface code and the application core are tangled. More specifically, the connection that brings both parts together turns out to be an obstacle when implementing such systems. Additionally, in current software engineering practices, different aspects of the user interface are implemented at different places throughout the application code. Hence user interface code and application code are tangled and scattered. In this dissertation we will consider the user interface code that implements the graphical interfaces of traditional business applications. These applications typically start from an application point of view (including the business behaviour) and a user interface is implemented on top of this. The provided user interface often consists of form-based windows with traditional widgets such as buttons, input fields, text labels, etc.

To illustrate the entanglement and scattering of user interface specific elements throughout the implementation, we present a small extract of code of the jGnash application. jGnash is a free cross-platform personal finance application written in Java [jGn]. The application itself is shown in Figure 1.1. The finance application contains several types of accounts such as investment, cash, liability, bank account, etc. The interface provides a front-end such that a user can select one of his accounts or create new ones. It is linked with the underlying application at several points. For example when clicking the button for creating a new account, the underlying application is informed of this request and creates the actual account for that particular user. Hence the clicking event for this button is linked with some action in the application. In its turn adding an account in the application will result in updating the displayed list of accounts. Hence this action in the application is linked with an action in the user interface which updates the display.

Apart from the interface and the application depending upon one another, also several dependencies exist between various parts in the interface itself. For instance, when selecting an account in part 1 of the interface, part 2 is updated such that it shows all the transactions for that account. Selecting one of these transactions shows its details in part 3 of the interface. Upon changing these details and re-entering the transaction, the current transaction is updated. Also interaction patterns exist, such as the CRUD (Create-Retrieve-Update-Delete) pattern. In a user interface such a pattern is for example represented by four buttons: new, edit, enter, delete. Clicking the new button results in a new entity (e.g. transaction), upon which it is not possible to edit or delete another transaction. Hence ideally these buttons are disabled when clicking the new button whereas the cancel button is enabled to return to the initial state. Similar enable and disable dependencies exist between the other buttons of the CRUD pattern. Obviously these dependencies can exist between any other user interface components as well. For instance when a new account of the type bank account is created, the UI shows a field for specifying the institution at which the account is held.

Figure 1.2 contains a small selection of code from the jGnash application. In this code we distinguish various concerns. The green code (dotted line border) is related



jGnash – 1.11.5 [/Users/sgoderis/finances.jgnash.xml]

MyBank Balance: €80.00 Reconciled Balance: €0.00

Date	Payee	Account	Deposit	Withdrawal	Balance
01/10/08	VUB	Bank Accounts	100.00		€100.00
01/15/08	supermarket	Bank Accounts		20.00	€80.00

Payee: VUB
Number:
Account: Bank Accounts
Date: 1/10/08
Memo:
Amount: 100.00
Reconciled
Exchange Rate:

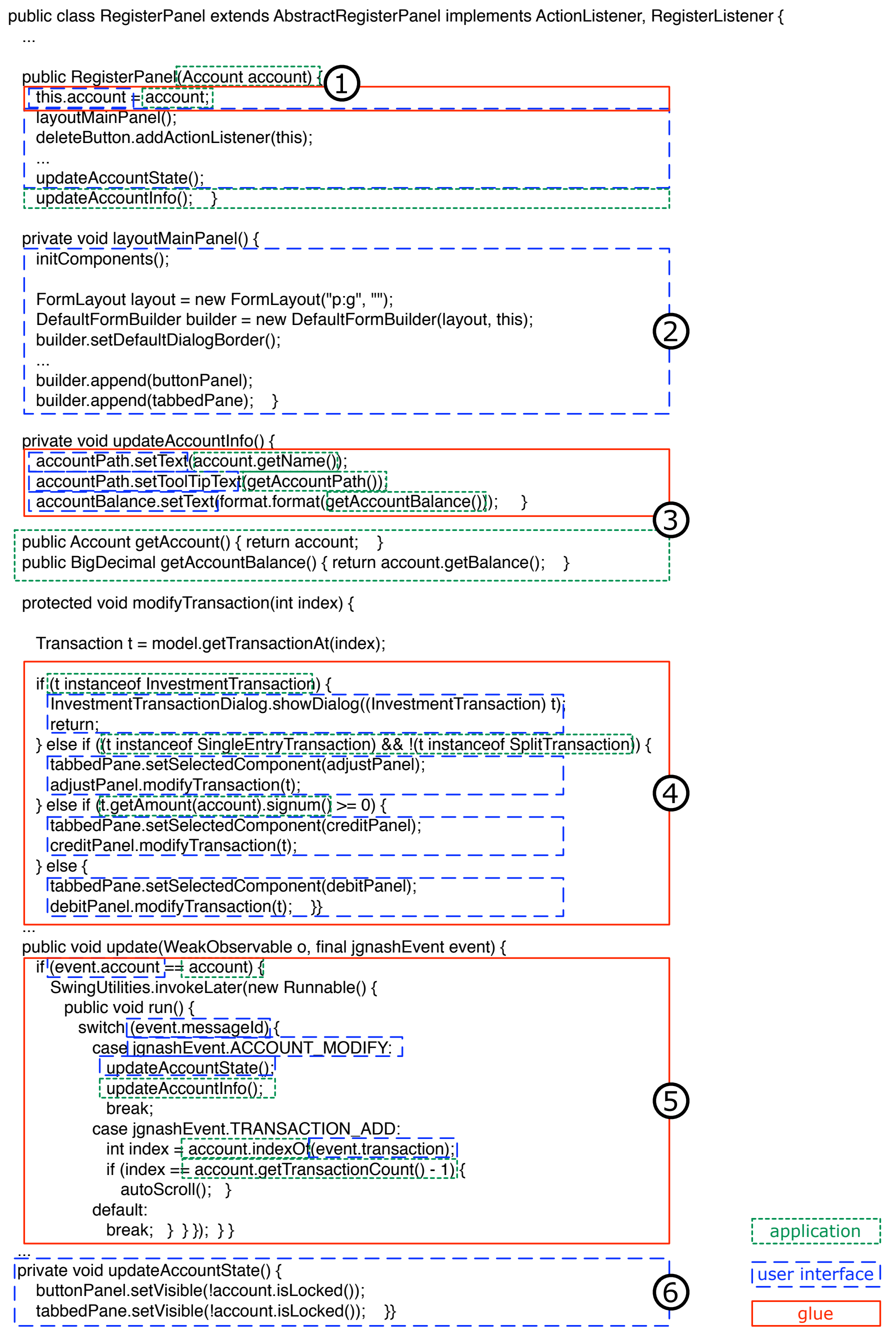
Transaction added
6958 Kb / 7244 Kb

Figure 1.1: The JGnash personal finance application

to the functional core of the application whereas the blue code (dashed line border) is concerned with user interface actions. These two parts are glued together with the red code (full line border) which specifies the control flow.

Looking at this code snippet shows that the various concerns occur at different places in the code file and are intertwined. The first phenomenon is called *code scattering*. This means that a single requirement affects multiple code modules [Tar99]. The second phenomenon is called *code tangling*. It occurs when code related to multiple other requirements is interleaved within a single module [Tar99]. Adding or evolving concerns that are scattered and entangled has a big impact on the application's code. A scattered concern implies an invasive change to different modules. Tangled code impedes comprehension and future evolution. Because of code scattering and tangling, making changes to one concern might break the functionality of another one.

The code snippet in Figure 1.2 is responsible for displaying the right hand dialogue (Figure 1.1 number 2 and 3) that corresponds to the selected account in the account list on the left hand side (Figure 1.1 number 1). For the selected account (Figure 1.2 number 1) a new layout is created (Figure 1.2 number 2) and the account info is updated (Figure 1.2 number 3). In order to update the account information, the actual account (application code) is queried and this information is passed on to the corresponding user interface elements. The second part of the snippet is related to showing the details of the selected transaction (Figure 1.2 number 4). When adding a new kind of transaction, this part needs to be updated in order for the interface to deal with this new kind. Note that the interface panes for these kinds of transactions (e.g. debitPanel)



```
public class RegisterPanel extends AbstractRegisterPanel implements ActionListener, RegisterListener {
  ...

  public RegisterPanel(Account account) {
    this.account = account;
    layoutMainPanel();
    deleteButton.addActionListener(this);
    ...
    updateAccountState();
    updateAccountInfo();    }

  private void layoutMainPanel() {
    initComponents();

    FormLayout layout = new FormLayout("p:g", "");
    DefaultFormBuilder builder = new DefaultFormBuilder(layout, this);
    builder.setDefaultDialogBorder();
    ...
    builder.append(buttonPanel);
    builder.append(tabbedPane);    }

  private void updateAccountInfo() {
    accountPath.setText(account.getName());
    accountPath.setToolTipText(getAccountPath());
    accountBalance.setText(format.format(getAccountBalance()));    }

  public Account getAccount() { return account;    }
  public BigDecimal getAccountBalance() { return account.getBalance();    }

  protected void modifyTransaction(int index) {

    Transaction t = model.getTransactionAt(index);

    if (t instanceof InvestmentTransaction) {
      InvestmentTransactionDialog.showDialog((InvestmentTransaction) t);
      return;
    } else if ((t instanceof SingleEntryTransaction) && !(t instanceof SplitTransaction)) {
      tabbedPane.setSelectedComponent(adjustPanel);
      adjustPanel.modifyTransaction(t);
    } else if (t.getAmount(account).signum() >= 0) {
      tabbedPane.setSelectedComponent(creditPanel);
      creditPanel.modifyTransaction(t);
    } else {
      tabbedPane.setSelectedComponent(debitPanel);
      debitPanel.modifyTransaction(t);    }}
  ...
  public void update(WeakObservable o, final jgnashEvent event) {
    if (event.account == account) {
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          switch (event.messageId) {
            case jgnashEvent.ACCOUNT_MODIFY:
              updateAccountState();
              updateAccountInfo();
              break;
            case jgnashEvent.TRANSACTION_ADD:
              int index = account.indexOf(event.transaction);
              if (index == account.getTransactionCount() - 1) {
                autoScroll();   }
            default:
              break;   } } }); } }
  ...
 private void updateAccountState() {
    buttonPanel.setVisible(!account.isLocked());
    tabbedPane.setVisible(!account.isLocked());    }}
```

Figure 1.2: An example of entangled and scattered user interface concerns

are implemented elsewhere. The last part of the snippet (Figure 1.2 number 5 and 6) is used to update the actual account (application code) and its display (interface code) upon certain account actions, for instance when a new transaction is added.

Furthermore if a new feature that is being added is context-dependent, it requires extra checks to know whether the code applies or not. The ability to permit arbitrary combinations of checks is also problematic and requires special infrastructure support, in both the design and implementation. This infrastructure usually comes at a high-cost in terms of conceptual complexity and runtime overhead [Tar99]. Currently new programming paradigms, like ContextL [Cos05, Hir08] provide such an infrastructure to deal with multiple contexts as an intrinsic part of their paradigm. How to incorporate the user interface concern into this paradigm is yet to be investigated.

The rise of importance for software variability, where applications need to be efficiently extended, changed, customised or configured, has consequences for the development of user interfaces. A programmer needs to be able to distinguish between the application code and the interface code and to understand how both are linked together. Changing either of these concerns should not lead to undesirable changes in another concern. Moreover implementing desired changes should not be needlessly cumbersome. Changing the application code possibly has an effect on the user interface code as new application behaviour might require the user interface to incorporate new features by changing its visualisation and its behaviour.

For example adding a new type of transaction to JGnash, requires the underlying application code to incorporate this new transaction type and its specific behaviour. The visualisation of the user interface changes as the new transaction will need an entry pane of its own, similar to the deposit and withdrawal pane. Also the user interface behaviour changes as making a different selection results in the new pane being displayed. Furthermore the user interface will call upon the application when submitting the new transaction. When code is scattered and tangled as in Figure 1.2 adding this new behaviour requires adaptations at several locations and invasive changes to control flow statements (e.g. number 4 in Figure 1.2). These control flow statements provide the glue between application and user interface. The developers have to deal with the complexity of these statements, especially when context changes are to be incorporated. Additionally, they also has to provide the code for dynamic user interface adaptations. In order to support the programmer in evolving and maintaining user interface code, application code, and the link between both, they should be able to deal with these three concerns in separation of each other. In the next section we elaborate on this separation for user interface concerns.

1.1 Separation of Concerns for User Interfaces

The principle of separation of concerns is well-known in software engineering research to achieve adaptability, maintainability and reusability in software systems [Par72]. The idea is that different kinds of concerns are involved in a software system. Focussing one's attention upon one concern at a time allows to cope with the complexity of a system. Hence, the different concerns are considered in 'isolation' of each other. By modularising the different concerns, separation of concerns deals with code tangling and scattering [Tar99]. With respect to separating user interface concerns, several approaches have considered applying the principle of separation of concern such that maintaining, evolving and reusing user interfaces is improved.

Current Approaches for Separation of Concerns in User Interfaces The best-know example of applying separation of concerns for user interfaces in current software engineering practices, is Model-View-Controller (MVC). The general perception of MVC is that separates the graphical interface (view) from the underlying application (model) by using an intermediator (controller) to handle the communication between the two. This controller usually provides a notification mechanism to propagate changes and is often implemented by using event handlers and value models. For instance this is the case for user interfaces implemented in .NET, Cocoa, Java and Visualworks Smalltalk.

Tiered architectures also distinguish between an application layer and a presentation (graphical user interface) layer. The application layer contains the business logic of the application and is called upon from within the presentation layer. Tiered architectures focus on an architectural separation. How to connect the various layers is left open for the developer of the architecture. This connection, although sometimes implemented in a dedicated layer of its own, is responsible for the communication and interaction between the presentation and application layer. Hence it will need to provide the necessary linking mechanisms, similarly to the controller mechanism in MVC.

Model-based user interface development (MB-UIDE) uses a set of different models in order to specify a user interface [Sch96]. These models consist of declarative specifications and represent all information about user interface characteristics that are required for the user interface development. Most model-based approaches use a generative approach to create the user interface based on these models. However, as MB-UIDE regained interest in the light of context-sensitive systems [Mye00], researchers have expressed the need for accessing the several user interface models at runtime [Dem06]. This is achieved by using techniques from model-driven engineering to provide for a mapping from model to runtime application. It is left open for the programmer to provide the actual link between application and user interface. Hence the programmer lacks support in specifying the complex control flows mentioned before.

Although separation of concerns is the driving principle behind aspect-oriented software development, aspect-oriented programming (AOP) has only partly been applied to user interfaces by implementing certain design patterns with aspects. The most popular implementation is the observer pattern which can be used to replace part of the manual linking mechanism used for MVC [Han02, Mil04]. In AOP the main functionality of the system is often implemented in an object-oriented programming language. Concerns that are cross-cutting this implementation are specified by using aspects, which are later woven through the main system automatically. When using aspects, one should keep in mind the obliviousness property as well as the fragile pointcut problem. At first AOP stated that the underlying base system should ideally be unaware of the aspects that are written on top, such that programmers do not have to expend any additional effort to make the AOP mechanism work. In AOP this is called *obliviousness* [Fil00]. However, this means that the aspects themselves specify when and where they are to be invoked. The pointcut definitions that are used to specify these points of invocation typically rely heavily on the structure of the base program and are therefore tightly coupled to the base programs structure [Kel06]. This gives rise to the *fragile pointcut problem*, which means that all the pointcuts of each aspect have to be checked and possibly revised whenever the base program evolves [Kop04]. When using AOP to achieve a separation of concerns, one should bear in mind the obliviousness and fragility of aspects and opt for an AOP approach that can deal with this, such as CARMA [Gyb03] and model-based pointcuts [Kel06].

Finally, declarative approaches and techniques have been applied to specify part of the user interface concerns. For instance Adobe uses declarative specifications for UI layout and for handling user events. JGoodies extends Java with declarative layout specifications. User interface description languages such as UIML, XUL and UsiXML, use XML-like specifications for the models used in MB-UIDE. Once more these specifications separate part of the user interface concern but lack support to aid the programmer in linking the underlying application with the user interface.

Limitations of Separation of Concerns in User Interfaces Although all these approaches provide some way to separate concerns in user interfaces, in practice software developers still need to be knowledgeable about both user interface and application in order to make both of them work together. The MVC controller for instance provides a notification mechanism, but instantiating the low-level mechanism often requires tangling and scattering both user interface and application code. Current software engineering solutions for creating user interfaces separate the application core from the interface, but neglect the different concerns within the user interface, which consists of the visualisation, the user interface behaviour and the link between user interface and application which still requires a programmatic intervention.

The link between user interface and application works both ways. Events in a user interface will trigger behaviour in the underlying application (a.k.a. call-back procedures). Web-based and form-based applications often address this link only. The same is true for tiered architectures. User interface events call upon the application to execute a certain action. Sometimes the direct result of this action is used to update the user interface, but rarely will an action in the application, that was not initiated by the user interface, have an effect on the user interface. Usually this is not even possible, as the application has no access to the user interface. However, other applications do require the user interface to be updated after application actions that were not initiated by the user interface. For instance this can be because an external application or another view on the same application triggered an action. At this point, the application needs to know about the user interface and the link between application and user interface

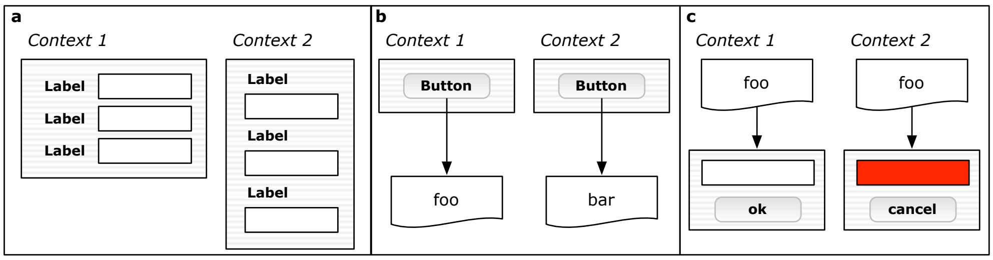


Figure 1.3: Context changes influence a) the user interface visualisation b) the link between user interface and application c) the link between application and user interface

becomes apparent. Note that here another view on the application means more than just having a different visualisation (graphical) possibility for the same user interface. The view can imply both a different visualisation and different user interface behaviour. Hence, two different views on the same application can act independently from one another but nevertheless trigger the same application, which is why the application should know what views depend on it in order to update them in case the application changed. This is according to the MVC meaning of using different views on the same application. *In short, both the link between user interface and application (call-backs) as the link between application and user interface are of importance. Whereas call-back procedures usually adhere to a separation of concerns, the mechanisms used to introduce the opposite link re-introduce code tangling and scattering.*

In the scope of context-sensitive devices, user interfaces typically need to respond to new context information and adapt accordingly. On the one hand this implies that application and user interface code becomes even more dependent on each other. Different contexts will require different application behaviour and user interface behaviour to become available. As illustrated in Figure 1.3 contexts can change the visualisation of a user interface (Figure 1.3 number a), but can also require the links between application and user interface to be updated (Figure 1.3 number b and c). Furthermore contexts will have to be combined. In context-oriented programming [Hir08] for instance, this is dealt with by plugging layers in and out. Yet context-oriented programming is not directly targeted at user interfaces. In other software engineering practices such behaviour results in complex control flow structures that are hard to maintain, especially when new context possibilities are added. On the other hand, user interface changes that are a result of changing context need to happen dynamically. This includes changing the user interface visualisation, such as updating component properties (e.g. visibility, colour, font, text) and their layout. Although several layout mechanisms (e.g. layout managers in Java) exist, advanced layout changes are hard to express and require the programmer to deal with complicated code, even for simple changes. User interface changes also include creating new links between the user interface and the underlying application, for example because clicking a button triggers different application behaviour in different contexts. Such a change would for instance require the event listener behind a button to be updated at runtime. To provide for such dynamism, the programmer should be supported when expressing this behaviour without having to deal with the technicalities of the underlying implementation mechanisms. Unfortunately current software engineering practices require the programmer to provide his own infrastructure. *In short, in current solutions dynamic user interface changes are driven by dedicated code that handles the programmatic reconfiguration of the interface components.*

When separating user interface concerns, the application developer and the user interface developer do not necessarily have to be the same person. This implies that a user interface designer can create the user interface without needing to know how it is linked with the underlying application core. The application developer can implement the application without having to consider what kinds of interfaces will be created and without needing to know how these interfaces will link with the application. Note that in both cases however some (possibly different) developer will need to specify how user interface and application will link together and thus provide a kind of connection between user interface and application. *At this moment, the latter is responsible for providing the necessary code and structures to link the application with the user interface and to allow for dynamic user interface changes.*

1.2 Towards Advanced Separation of Concerns for User Interfaces

The goal of the approach we propose in this dissertation is to achieve an improved separation of concerns for user interfaces. In order to do so, we first need to consider what concerns to actually separate. Existing approaches focus on separating the user interface and the application from one another, thereby overlooking the entanglement that appears in the mechanism that links user interface and application. The user interface and application are linked because an event in one can trigger an action in the other, but also because different choices (or contexts) at the application level require different actions at the user interface level.

Figure 1.4 shows the various user interface elements we envision. The application core is considered to be developed in complete obliviousness of user interfaces and we do not consider it to be a user interface concern. However, how an application is triggered from within the user interface and how the user interface is triggered from within the application, is an important concern. In this dissertation we refer to it as the **application logic concern**. It consists of application actions and application events. *Application actions* refer to application behaviour that is called from within the user interface. *Application events* trigger the user interface from within the application. Another concern, the **presentation logic concern**, is related to the user interface itself and specifies the visualisation concern (i.e. what the interface looks like) and the behavioural concern (i.e. how the interface behaves). The latter consists of user interface events and

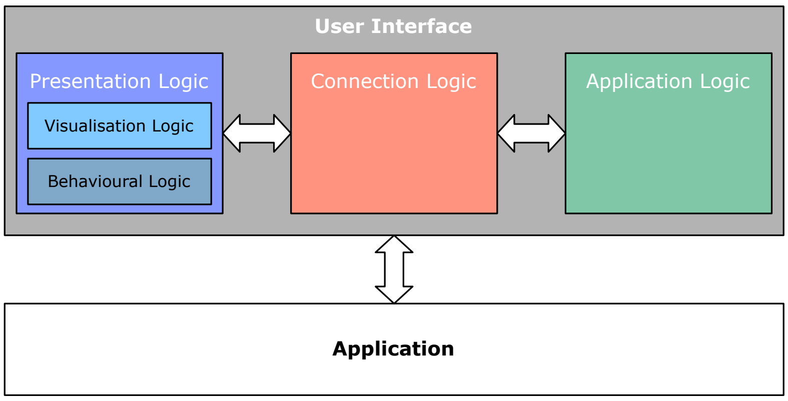


Figure 1.4: The different elements in a software system

user interface actions. *User interface events* are user operations that will trigger either user interface or application behaviour. *User interface actions* refer to user interface behaviour that changes the user interface. Note that the user interface logic concern is completely unaware of how to interact with the application. As both application and user interface events can trigger both application and user interface actions, both the application logic and the user interface logic concern need to be connected with one another. This is what we call the **connection logic concern**. It is responsible for linking events to actions. On the one hand, a user interface event can call the underlying application core to execute the corresponding application behaviour. On the other hand the application and user interface events can call upon the user interface logic for updating its state and visualisation.

Now that we know what concerns to deal with, there are several requirements to be kept in mind when providing the programmer with the necessary support for applying a separation of user interface concerns such that the tangling and scattering we introduced earlier is avoided.

First, applying the principle to the three concerns mentioned above signifies that each of these concerns is *specified in separation* from the others. Remember that the ultimate situation we envision is for a programmer to be able to work on one of the concerns without having to worry about the others.

Second, the programmers' understanding of the various concerns is improved by introducing *levels of abstraction*. For example when creating an interface, the user interface designer needs to know what set of widgets are available and what set of property changes are allowed (e.g. disable, change text, close window). The do not need to know (and remember) how an actual widget is implemented or how a property is actually changed.

Third, the different levels of abstraction should be *mapped* onto one another *automatically*. Once such a mapping is provided, either the low-level or the high-level specifications can be changed without affecting the other. For instance the low-level (implementation) can be changed from a set of Smalltalk widgets to a set of HTML widgets without requiring changes to the high-level user interface which was created by the user interface designer. By automatically applying the new mapping, the interface for the different platform is created.

Fourth, it is obvious that an application and its interface should cooperate. Therefore they are linked with one another, but the mechanism to achieve this linking breaks the separation and requires the concerns to be aware of each other. In current software engineering approaches putting the linking mechanism in place requires the programmer to entangle the code for the various concerns and to know about the low-level implementation of each of these concerns. Linking user interface and application becomes a cumbersome task. In order to maintain the separation of concerns from a programmer's point of view, the *linking mechanism should be put in place automatically* and manually fine-tuning the code afterwards should be excluded.

Finally, some of the current software engineering approaches have acknowledged the difficulty of creating a 'good-looking' user interface layout. As a user interface designer benefits from high-level specifications, this is also true for specifying layout. Ideally layout is described with abstract relations which are transformed into an actual *layout automatically*. On top of this, layout strategies will allow for abstract layout specifications which can apply for different user interfaces. For instance, layout guidelines imposed by a company will apply to all the interfaces used in that company, while the same user interface specifications with different guidelines can be used for a different company.

In addition, when providing a solution that supports the programmer in separating user interface concerns, there are several considerations to keep in mind. Firstly, we see a recurrent tendency towards using declarative style mechanisms to describe user interfaces. For example, Adobe's Adam&Eve and UIML, have moved towards declarative specifications for describing what a user interface looks like. We also approve of this and believe that a *declarative medium* to describe the various concerns is the way to go. It allows us to focus on the *what* instead of the *how* (or the process of assembling) of user interface concerns.

Secondly, as the different concerns need to be combined into a working software system automatically, a mechanism should be provided to do so. Implementing this mechanism will benefit from having the various concerns expressed in a uniform medium, or at least transformed into a specification using the same *uniform medium*. Such a uniform medium will simplify the assembly process.

In this dissertation we propose DEUCE[1], an instantiation for a solution that takes the above mentioned requirements and considerations into mind. DEUCE uses a declarative language to specify the different user interface concerns declaratively and in a uniform medium. The medium supports specifying the concerns in separation from one another and at different levels of abstraction. The reasoning mechanism provided by the declara-

[1]DEUCE stands for "DEclarative User interface Concerns Extrication".

tive programming language is used to combine the various concerns with the underlying application, and hence provide the final running software system. Furthermore it allows for runtime user interface changes.

Although DEUCE does not use traditional aspect-oriented techniques, it is to be situated in the realm of Aspect-Oriented Software Development. The various user interface concerns are specified in isolation from the application. Part of the specification indicates how these concerns are linked with the application and DEUCE provides the mechanisms to compose the concerns and the application into a final running system. As such, DEUCE uses a kind of aspects, pointcuts and a weaver in order to modularise user interface code.

1.3 Contributions

In this dissertation we will address the following conceptual and technical contributions:

- We provide an accurate definition of terminology for the several UI concerns, namely the presentation, the application and the connection logic concern.
- We propose a conceptual solution for achieving a separation of user interface concerns. This solution is explained by postulating five requirements.
- We provide the proof-of-concept implementation DEUCE as an instantiation of the conceptual solution. In this solution, we use a declarative language to specify the several concerns. These specifications are combined with the underlying application into a running software system. To do so, DEUCE uses a declarative reasoning engine and meta-programming mechanisms such as method-wrappers and reflection. DEUCE offers support for creating VisualWorks Smalltalk user interfaces by abstracting away from the low-level implementation technicalities towards high-level expressions.

1.4 Outline of the Dissertation

This dissertation is structured as follows:

- **Chapter 2: Separation of Concerns in User Interfaces**
  Several approaches towards a separation of concerns in user interfaces exist. We will address especially the Model-View-Controller (MVC) metaphor as it is still omnipresent in today's software engineering practices when it comes to separating the concerns from an implementation point of view. In this chapter we will discuss the general aspects of MVC and show how it is applied in several of those software engineering practices. Furthermore we discuss the use of aspect-oriented programming for separating user interface concerns as this is a well-known modularisation technique. Also model-based UIs and other declarative approaches are discussed in this chapter.

- **Chapter 3: A Foundation for Separating User Interface Concerns**
  In this chapter we will first analyse the existing work for separating user interface concerns and explain why these solutions are insufficient. Next we will provide a refined terminology with respect to concerns in user interfaces, which will be used throughout the rest of this dissertation. The conceptual approach we propose in this dissertation for achieving an improved separation of user interface concerns will be discussed together with the requirements such an approach should adhere to.

- **Chapter 4: DEUCE: A Proof-of-concept for Separating User Interface Concerns**
  We will instantiate the conceptual approach by proposing a practical implementation in the form of DEUCE. DEUCE uses a declarative language for expressing user interface concerns. In this chapter we will explain the choice for a declarative approach as well as the language SOUL we have used to provide for this approach. We will end this chapter by illustrating the functionality of DEUCE from a developer's point of view.

- **Chapter 5: The Internals of DEUCE**
  In this chapter we will introduce the mechanisms that are used to implement DEUCE and provide the necessary functionality of our solution. DEUCE uses SOUL's reasoning engine and its symbiosis with VisualWorks Smalltalk (and hence its reflective capabilities) in order to create user interfaces and their link with the application. The VisualWorks Smalltalk UI framework is accessed from within SOUL in order to construct a user interface and to link the UI with DEUCE. The application is linked with DEUCE through the use of method-wrappers. The actual link between application and UI (and vice versa) is expressed at the logic level. We will also discuss the Cassowary constraint solver which is used for automatically laying out the UI.

- **Chapter 6: Validation with *MijnGeld***
  We will validate DEUCE by showing how it supports the creation, reuse and evolution of the *MijnGeld* application. This personal finance application makes a clear distinction between core finance functionality and its user interfaces. Through the use of several scenarios we will show how UI specifications can be reused and how the several concerns can be evolved in separation of one another.

- **Chapter 7: Conclusion and Future Work**
  We will conclude this thesis with a recapitulation and some future work. We will discuss some of the technical improvements we see for DEUCE as well as some future research directions.

Chapter 2

Separation of Concerns in User Interfaces

By modularising the different concerns involved in a software system, these concerns can be considered in isolation from one another. This allows developers to focus their attention on one concern at a time and therefore better cope with the complexity of software systems. This principle of modularising concerns is called 'separation of concerns'. In general it is acknowledged as an important step to increase the adaptability, maintainability and reusability of software systems [Par72]. On a practical level it is known to reduce code tangling and code scattering [Tar99]. Code tangling refers to code for different concerns being intertwined and typically occurs when code with respect to multiple other requirements is interleaved within a single module. In addition the developers' understanding of the system is obfuscated. Evolving one of the concerns requires developers to have a thorough understanding of what code is related to what concern. Furthermore they have to make sure that changing the concern does not undesirably affect the other concerns. Code scattering means that code for a certain concern is located at different places throughout the software system. This means that a single requirement affects multiple code modules. Hence, evolving or adding a concern implies an invasive change to different modules. Scattered and tangled concerns are also known as cross-cutting concerns.

The principle of separation of concerns has been applied to separate user interfaces from the underlying application. One of the best known approaches is the Model-View-Controller metaphor, which is still applied in today's popular software development platforms. In this chapter we discuss the general aspects of this metaphor in Section 2.1 and how it is applied in several of those software development platforms in Section 2.2. In Section 2.3 we discuss other approaches for separation of concerns in UIs, namely multi-tiered architectures, model-based UIs, aspect-oriented programming and declarative UI approaches and techniques.

2.1 Model-View-Controller

The original Model-View-Controller pattern (MVC) was first conceived in 1979 by Trygve Reenskaug [Ree79b]. It became popular through its implementation in Small-

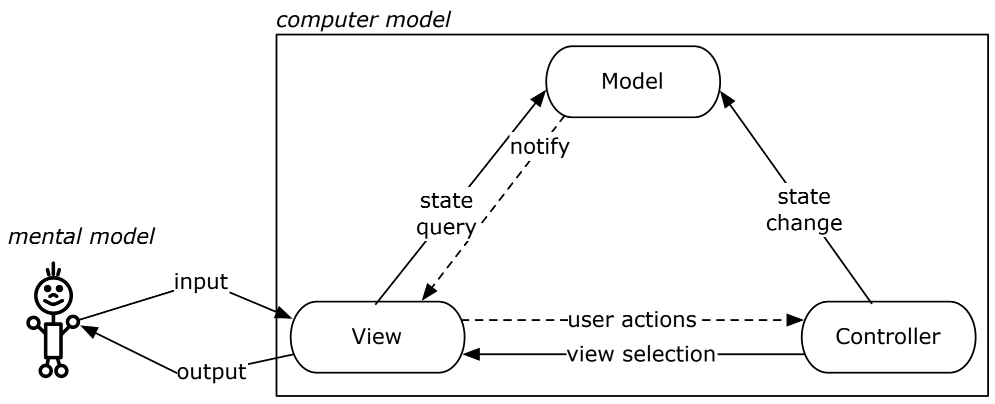


Figure 2.1: Model-View-Controller

talk-80, as described by Krasner and Pope [Kra88]. Although MVC has been around for a long time it still plays an important role for separating concerns in user interfaces in today's software development. The MVC pattern has had different interpretations over the years. The original pattern [Ree79b] evolved, several other patterns like the observer pattern were distilled [Gam95] from it, and sometimes typical implementations, such as the Visualworks Smalltalk implementation [Hun97], were mistaken for the actual pattern.

In this section we first introduce the MVC concepts as they are generally perceived today. Next we elaborate on two of the techniques used to implement the controller part of MVC. Finally we cover some other MVC-alike models.

2.1.1 MVC Concepts

Figure 2.1 shows the concepts of the architecture of the MVC metaphor as it is generally perceived today. As is shown, the main components are model, view and controller.

Model The user of a software system has a certain mental model of the world that is represented by a computer *model* in the system. This model represents the application's domain-specific knowledge and contains the components of the system that do the actual work [Kra88]. This is for example, the core application behaviour of a personal finance system with accounts, customers, transactions, etc.

View Depending on the task being performed by the user the model can be looked at from different perspectives. Therefore the system should offer different views on the model. Such a *view* acts as a (visual) presentation filter on the model by highlighting certain attributes and by suppressing others [Kra88]. For example the view for a novice user of a system can be limited and leave out behaviour that is available to an administrator. In the personal finance application for instance, one view shows a listing of all transactions for a certain account, while another view shows a pie chart for the categories used in those transactions. Both views use the same underlying application data, namely the transactions, but show a different presentation.

Controller Originally the *controller* was intended to act as an interface between the model with its associated views on the one hand and the interactive user interface devices (e.g. keyboard and mouse) on the other hand [Kra88]. In the contemporary interpretation it is thought of as the intermediary between the model and the view [Fow06]. This makes the view responsible for dealing with devices in- and output. Hence the controller is responsible for propagating state changes and actions in the user interface to the application, and vice versa. For example in the personal finance application adding a transaction in the user interface is reflected in the underlying application data. Additionally, changing the underlying application data, for example by loading an external file with financial transactions which adds these transactions to the underlying data, will in its turn have an impact on the user interface.

Interactions between Model, View, and Controller The controller component in MVC uses a notification mechanism to assure that changes in the model are reflected in the view, and vice versa. As the values of UI components change (e.g. through user input), this might change data in the model. Additionally, UI actions (e.g. user events) will trigger certain behaviour of the model, which in its turn might also imply data changes in the model. When a change in the model is to be reflected in the UI, it should be reflected in all the model's views, and not just the view that initiated the change. The model will thereby notify all its views, which in their turn can query the model for information and update themselves if necessary. This change-notification mechanism, a.k.a. observer or publish-subscribe [Gam95], is used to keep model and view decoupled.

2.1.2 Notification Mechanism

As mentioned above, MVC uses a notification mechanism to link view and model together. Hence this mechanism plays a key role to achieve a separation of concerns in user interfaces. The notification mechanism is implemented either through an event handling system, a data binding mechanism, the observer pattern, or a combination thereof. An event handling system can be used to link both UI state and actions to the model. A data binding mechanism can be used to link the UI state to the model's data. It is used in combination with an event system that links the UI actions to model behaviour. The observer pattern is typically used to link changes in the model back to the UI. Section 2.2 illustrates how these implementation mechanisms are put into practice.

In *event handling*, events and event handlers are used to make an application respond to either user or other events. For example, if a user clicks the send button in an email application, the application makes sure it gets sent. An event is a message sent by an object to signal the occurrence of an action which is caused by a user interaction or triggered by some program logic. The object that triggers the event is called the event sender. The object that captures the event and responds to it is called the event receiver. The event handler is the method executing the program logic in response to the event. Registering the event handler with the event source is referred to as event wiring [MSDa].

Frameworks implementing MVC typically use a *data binding mechanism* to establish a connection between UI components and the application logic. When the data changes its value, the elements bound to the data automatically reflect changes. Vice versa, if the representation of the data (i.e. the UI component representing the value) changes, the underlying data in the application is also automatically updated.

The *observer design pattern* [Gam95] is used to decouple the model from the view in such a way that the model does not know about the views explicitly but nevertheless is able to inform the views of changes. This allows for both the model and the view to be reused independently but nevertheless to work together. Several views can depend on one model, without the model knowing about the view but behave as it would know. When the user changes the model, the views will be notified of this change and respond accordingly. The key objects in the observer pattern are subject and observer. A subject (model) may have any number of dependent observers (views). Whenever the subject undergoes a change in state, all observers are notified. In response, each observer will query the subject to synchronise its state with the subject's state.

2.1.3 Other MVC-like Models

Over the years several models similar to MVC have been proposed to achieve a separation of concerns in UIs, such as the Seeheim model, the Arch model and Model-View-Presenter. The different components in these models are illustrated in Figure 2.2.

Seeheim

The Seeheim Model (Figure 2.2a) can be seen as the basic model [Eve99] for UI management systems, upon which MVC is also based. It divides user interfaces into three components: presentation, dialogue controller and application interface. The application is the functional core of the system and models the domain-specific concepts [Cou93].

The application interface handles the communication between the user interface and the back-end application. Hence, it represents the system from the viewpoint of the user interface application and contains all data structures and functions that belong to the underlying application that should be available to the user interface application. Furthermore it analyses the data before they are sent from the user interface to the non-user interface application [Car93].

The presentation component is responsible for the external representation of the user interface. It provides a mapping between this representation and the internal, abstract representation of input and output [Eve99]. It controls the end user interactions [Kaz93].

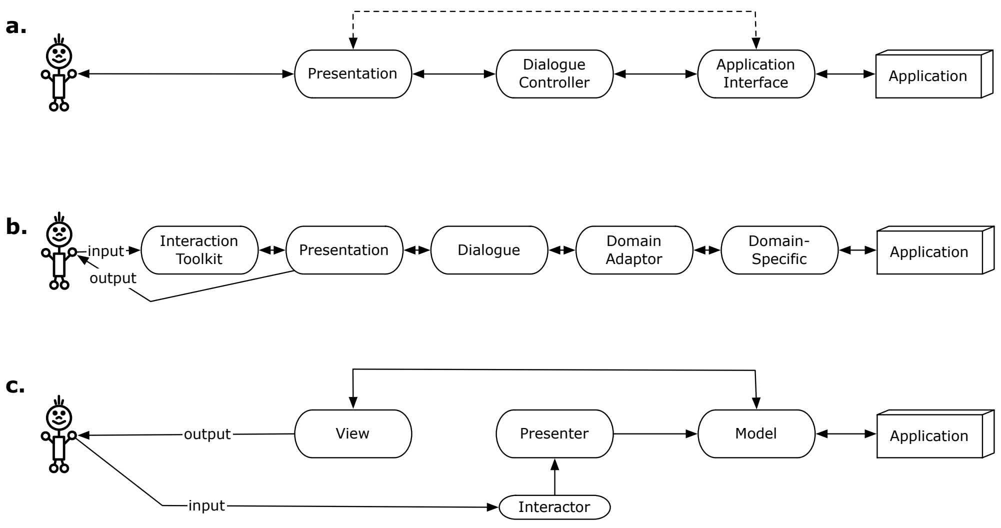


Figure 2.2: User interface models: a) Seeheim b) Arch c) Model-View-Presenter

The dialogue controller defines the structure of the user-computer dialogue. It is responsible for maintaining a correspondence between the application domain and the presentation domain. In the original Seeheim model, the dialogue controller contained the state of the user interface application [Eve99]. Other interpretations refer to it as a mediator between the states of the presentation and the application interface component [Car93, Kaz93]. The bridge between the presentation and application interface bypasses the dialogue controller for the flow of large amounts of data [Eve99, Kaz93].

The model, view and controller components in MVC as it is perceived today map onto the three components in Seeheim. In this perception the controller maps onto the dialogue controller and is used as a mediator between the view (presentation) and the model (application interface).

Arch

The Arch model (Figure 2.2b) is a refinement of the Seeheim model [She92]. It provides a model of the runtime architecture of an interactive system. Both the domain functionality and the user interface toolkit are taken into account as both are perceived as constraints on the development of a user interface [Bas92]. The domain specific and the interaction toolkit components form the basis of the architecture. Other essential functionality is provided by the dialogue, presentation and domain adaptor components.

The domain specific component corresponds to the application in the Seeheim model [Cou93] and it provides the functional core of the system such as data manipulation and other domain oriented functions [Kaz93]. The domain adaptor component corresponds to Seeheim's application interface and acts as a mediatior between the domain specific and dialogue component. It implements domain-related tasks that are required for human operation of the system but that are not available in the domain specific component [Bas92]. The combination of the domain specific component and the domain adaptor component maps onto the model in the MVC metaphor.

The presentation component in Seeheim, and thus the view component in MVC, corresponds to the interface toolkit together with the presentation component in Arch [Cou93]. The interface toolkit component implements the physical interaction with the end-user. The presentation component acts as a mediation component between the interface toolkit and the dialogue component and provides a set of toolkit independent objects that can be used by the dialogue component. Also decisions about representation are made here [Bas92].

The dialogue component, in analogy with the dialogue in Seeheim [Cou93], is responsible for task-level sequencing, providing multiple view consistency and for mapping domainspecific formalisms with user interface specific formalisms [Bas92]. Just like the controller in MVC it mediates between the domain and the presentation components.

Model-View-Presenter

Model View Presenter (Figure 2.2c) is a generalisation of the MVC metaphor and was conceived at Taligent in the 1990s [Pot96]. Dolphin Smalltalk further popularised the idea and used MVP to overcome some of their problems with MVC as it is implemented in VisualWorks Smalltalk [Bow00]. Note that in this implementation the controller was originally used to deal with user input and output rather than as a mediator between model and view. The MVP generalisation approaches the current perception of MVC.

In MVP the model and view have the same meaning as in MVC. So the model is the data the user interface will operate on and the view displays the content of the model. MVP [Pot96] adds abstractions for specifying subsets in the model's data and abstractions for representing the operations that can be performed on this data. The controller has been replaced by interactors. Interactors capture events on the user interface. The presenter interprets these events and provides the business logic to map them onto the appropriate commands for manipulating the model [Pot96, Fow06]. The degree to which the presenter controls the widgets in the view varies [Fow06]. In [Pot96] the presenter does not get a say in how the view logic is handled and all this logic stays within the view. In [Bow00] however, the presenter handles the view logic in more complex cases. This way the behavioural complexity gets extracted from the basic widget which makes it easier to understand. However, a major drawback is that the presenter component is closely coupled with the screen, therefore annihilating the reason for decoupling in the first place.

Morphic and PAC

Morphic and PAC are two models often mentioned alongside with MVC, but they do not provide a separation of user interface concerns as such.

Several Smalltalk environments exist and while some of these, like Visualworks and Dolphin, implement MVC or a variation thereof, others provide a different approach.

Squeak for instance uses morphic as a direct-manipulation user interface construction kit. However, the graphical objects in morphic (called morphs) combine the view and controller aspect (as used in MVC) into one object, and often also include the model aspect. Therefore these three concerns are combined, and strongly entangled, for each of these entities.

The same is true for the Presentation-Abstraction-Control (PAC) model. Its abstractions are similar to MVC, but in this model each UI component is represented by a PAC agent. Therefore each UI component contains all three facets, analogous to morphic. In PAC-Amodeus, PAC is integrated with the Arch model by applying its principles to the dialogue component [Cou97, Kaz93]. Here the abstraction facets define the internal state of the interaction, the presentation facets define the external state of the interaction, and the control facets define the mapping functions between internal and external state [Nig91].

2.1.4 MVC for Separating Concerns in User Interfaces

The main drive behind the model-view-controller metaphor is to separate the user interface from the application such that it is possible to provide multiple views for the same application. This implies that the application might change through other channels than the one provided from a certain view. Therefore it is the application that has the responsibility of notifying its views upon a change. Although such notifications are supported by a notification mechanism, in practice *the developers remain responsible for triggering this notification mechanism.* For example, upon updating a data field in the application, the notification mechanism has to be notified that some change happened such that it can propagate this notification to the dependent views. As a result, developers are still confronted with the low-level implementation of the notification mechanism. Taken into account that the notify message is part of the UI concerns (it is only relevant within the context of having UIs), these *UI concerns are still tangled and scattered throughout the application code.*

In addition, MVC separates the view from the application but neglects the code tangling in the view itself. For example, updating the user interface in different ways for different contexts, introduces *complex control flows which will result in adhoc mechanisms* being implemented by the developers themselves in order to deal with this complexity. Furthermore context changes usually require *dynamic UI changes which are taken care of programmatically* and for which MVC in general also does not support developers.

2.2 The MVC Metaphor Put into Practice

The Model-View-Controller metaphor is widespread in today's software development platforms. The differences between the several platforms mainly lie in how they implement the controller's notification mechanism. In what follows we will give the example of .NET, Cocoa, Java and Visualworks Smalltalk.

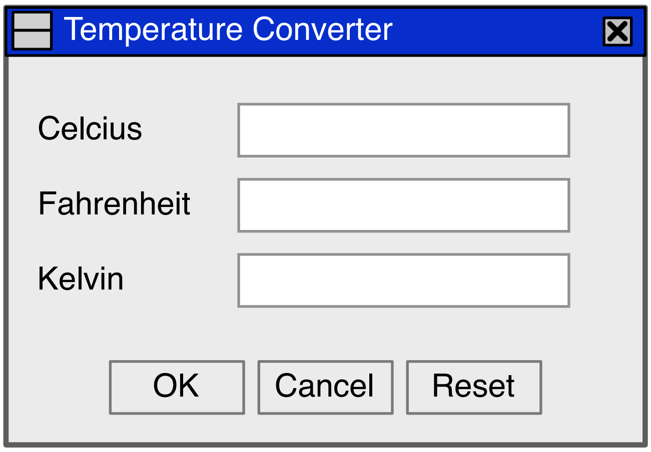


Figure 2.3: The temperature converter user interface

Note that many of the popular software development platforms are supported in an industrial context rather than an academic context. A lot of documentation is available through websites and tutorials and a large number of practical examples is freely available on-line. Throughout this section we will use the temperature converter UI depicted in Figure 2.3.

2.2.1 User Interfaces in .NET

.NET is a popular development environment for creating applications for the Windows platform. It is a framework for developing Microsoft applications which support several programming languages, including C# and Visual Basic .NET [Dot]. What language is used makes little difference as they are merely considered to be a different syntax to the same .NET common class library.

A programmer can develop two kinds of applications with .NET, namely window or web applications. User interfaces for the first use winForms, whereas .ASP pages are used for the second kind of applications. Both use event driven programming for linking the UI with the application [Dot]. The events are used to trigger actions upon UI events as well as for informing the application of state changes in the UI.

Event Handling

Forms can be created by using the Windows Forms Designer in .NET Visual Studio. This tool will generate event handler stubs automatically for default events. The actual event handling code is to be implemented by the developers [Dev]. This event handling code will be *responsible for updating the UI (programmatically)* when dynamic UI changes are required, for instance because of a context change. After implementing this event handling code one is limited to using the tool to update the UI, as *re-generating will result in a loss* of the manually added implementation.

In user interfaces, event senders are often user interface controls such as buttons, windows, etc. These event senders are responsible for triggering an event. Events are dealt with by event handlers. Hence, these contain the code to execute upon receiving the event. In order for event senders to know what handlers should be triggered, an event handler registers itself with the event. The event sender will notify all registered handlers upon events without knowing about any of their specifics [MSDa].

The following code example shows how events are used in Visual Basic .NET. The sender class `MyButton` contains an event declaration for the `MyClick` event (lines 1–2) and the `onMyClick` method that raises the event (lines 3–4). The receiver class indicates that it understands events thrown by `MyButton` through the use of the `withEvents` keyword (line 8). It also provides an event handler to deal with the `MyClick` event (lines 17–20), designated by the use of the `Handles` keyword (line 18). Alternatively, we can use the standard `Click` event provided by the .NET `Button` class (lines 11–16).

```
 1  Public Class MyButton
 2  Public Event MyClick(ByVal s As Object, ByVal e As System.EventArgs)
 3  Protected Overridable Sub onMyClick(ByVal e As System.EventArgs)
 4  RaiseEvent MyClick(Me, e)
 5  End Sub
 6  End Class

 7  Public Class TemperatureConverterActions
 8  Friend WithEvents okButton As MyButton
 9  Friend WithEvents cancelButton As System.Windows.Forms.Button
10  Friend WithEvents resetButton As System.Windows.Forms.Button

11  Private Sub Reset(ByVal s As System.Object,ByVal e As System.EventArgs)_
12  Handles cancelButton.Click, resetButton.Click
13  CelciusTextBox.Text = ""
14  FahrenheitTextBox.Text = ""
15  KelvinTextBox.Text = ""
16  End Sub

17  Private Sub Convert1(ByVal s As System.Object,_
18  ByVal e As System.EventArgs) Handles okButton.MyClick
19  convertTemperatureCelciusToFahrenheit()
20  End Sub

21  Private Sub Convert2(ByVal s As System.Object,_
22  ByVal e As System.EventArgs) Handles okButton.MyClick
23  convertTemperatureCelciusToKelvin()
24  End Sub
25  End Class
```

The `withEvents` and `Handles` keywords are the two parts that wire up a specific event handling procedure with the object launching the event. In addition to using these keywords, .NET provides a way to add and remove handlers with the `AddHandler` and `RemoveHandler` statements. These can be used at runtime to dynamically connect an object's events to event procedures. Hence developers can provide the necessary code to make these event handlers available at different times or different contexts [Bre03, Dev].

The event handling system in Visual Basic .NET allows for handling multiple events by a single event handler as well as for handling a single event by multiple handlers [Dev]. For instance in the example above the event handler `Reset` handles both the click event sent by `resetButton` and `cancelButton` (line 12). Having multiple events handled by a single event handler allows the code for one functionality to be concentrated at one place, even if it is to be triggered by multiple events. Making adaptations to this functionality means these have to be done at this location only. On the other hand the *developers need to provide testing code* if they want to determine what event triggered the handler.

The event `okButton.Click` is handled by both the event handlers `Convert1` (line 18) and `Convert2` (line 22). Hence, upon clicking the `okButton` both these handlers will be executed. However, the order in which the events are handled is not defined. Writing multiple handlers for one event allows us to add context checks such that event handlers depend on the context and are available at different times or in different scopes. With this added flexibility comes added complexity as *code related to one event gets scattered.* When changing or deleting the event, the developers need to scan all the code to find the necessary places for adaptation.

User Interface Process Application Block

In addition to the event handling system discussed above, the User Interface Process (UIP) Application Block is a Microsoft pattern that was introduced in .NET to simplify the development of user interface processes. It is designed to abstract the control flow (navigation) and state management out of the user interface layer into a user interface process layer. To do so it uses the principles of MVC, but note that the *UIP Application Block does not separate application model and interface*, but the user interface process from the graphical view. Its model is used to store user information and control information within the user interface process. Its controller is responsible for the starting of, navigating within, and ending of a user interface process [MSDb, MSDc]. Its view is still the visual interface.

2.2.2 User Interfaces with Cocoa

Cocoa is an object-oriented application environment designed for developing Mac OS X native applications and is gradually becoming the norm for Mac OS X [App06]. The preferred applications to use for developing Cocoa software are Xcode and Interface Builder. The latter is a graphical tool for creating user interfaces. Through its palettes, user interface objects can be dragged onto the appropriate surfaces (e.g. a window).

Its inspector can be used to set the initial attributes and sizes of these objects as well as to establish connections between objects [App06]. Cocoa offers several mechanisms to make objects communicate with each other, and hence also to connect UI objects with application objects. One mechanism uses outlet and target-action connections. Outlet connections link UI data with the application, whereas target-action connections link UI actions with application actions. Targets and actions are usually set through the Interface Builder, although they can also be changed programmatically. Both a delegate mechanism and notification mechanism are provided to propagate changes [App06] when using outlet and target-action connections.

More recently the Cocoa bindings mechanism was introduced to replace the above (traditional) Cocoa mechanisms. Bindings provide a collection of technologies to fully implement a model-view-controller paradigm and reduce the glue code and code dependencies between models, views and controllers [App07]. As this implies a better separation of concerns for user interfaces, we will only discuss the bindings mechanism in more detail.

Data Binding: Cocoa Bindings

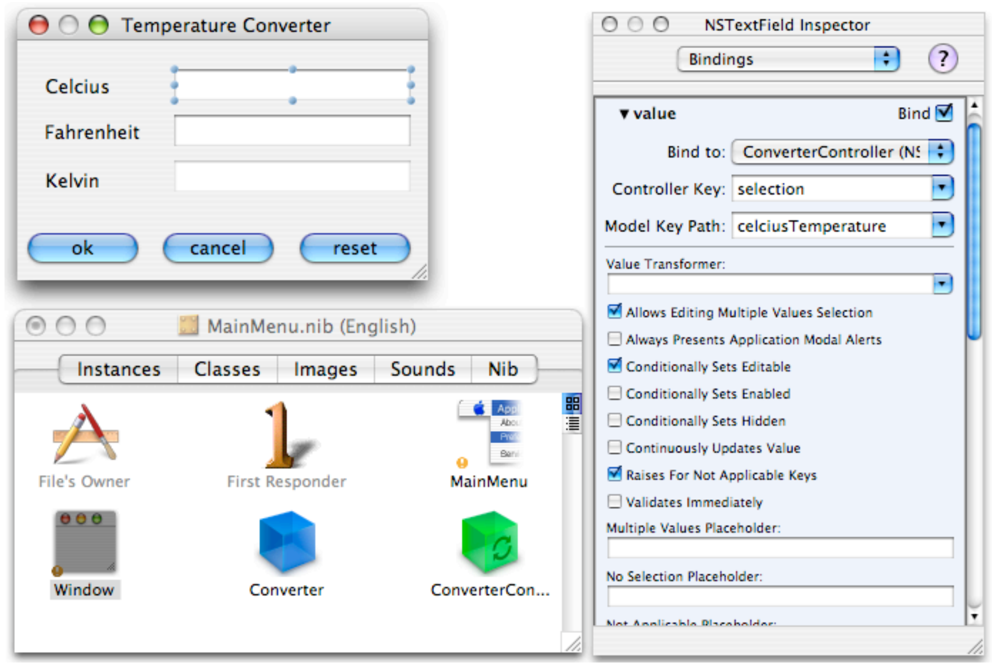
Bindings synchronise the display and storage of data in an application. It is an adaptation of the model-view-controller pattern that automatically puts observer behaviour in place. Bindings establish a mediated connection between the attribute of a view object that displays a value and a model object property that stores that value. Changes in one side of the connection are automatically reflected in the other [App06, App07].

Aside from adding view and model objects, the developers also provide controller objects to act as an intermediary. This controller object uses some fundamental parts of Cocoa, namely key-value binding, key-value coding and key-value observing. These allow objects in an application to observe changes in the properties (either attributes or relationships) of other objects. However, it is important that properties of objects are compliant with the requirements for key-value coding. This means certain naming conventions and return type conventions are to be followed when specifying instance variables, accessor methods and validation methods. The Interface Builder can be used to apply bindings between objects and set up this key-value mechanism. It is also possible to introduce or change bindings manually.

Key-value binding establishes the bindings between objects and also removes and advertises those bindings. Figure 2.4 shows an example where the Interface Builder is used to insert a binding between the `celciusTemperatureField` and `ConverterController`. This binding corresponds to the following piece of code:

```
[celciusTemperatureField bind: @"value" toObject: ConverterController
                withKeyPath:@"selection.celciusTemperature" options:nil];
```

With this binding the value of the `celciusTemperatureField` is bound to the `celciusTemperature` of the object to which the `ConverterController` is pointing at, namely its `selection`. The `ConverterController` acts as a go-between between the



▾ value Bind ☑

Bind to: ConverterController (NS

Controller Key: selection

Model Key Path: celciusTemperature

Value Transformer:

☑ Allows Editing Multiple Values Selection
☐ Always Presents Application Modal Alerts
☑ Conditionally Sets Editable
☐ Conditionally Sets Enabled
☐ Conditionally Sets Hidden
☐ Continuously Updates Value
☑ Raises For Not Applicable Keys
☐ Validates Immediately

Multiple Values Placeholder:

No Selection Placeholder:

Figure 2.4: Bindings in Cocoa interface builder

temperature inputfield of the UI and the actual temperature converter application. This binding code provides the connection between application and UI. As it can be expressed programmatically as well, these statements can be used to provide context dependent control flows that decide what binding to apply or to change. Nevertheless, these control flow statements have to be implemented by the developers and *entangle application knowledge with UI knowledge.*

Key-value coding makes it possible to get and set the values of an object's properties through keys without having to invoke that object's accessor methods directly. For example, accessing the value of the `celciusTemperature` property, is achieved by:

```
[Converter valueForKey:@"celciusTemperature"]
```

Key-value observing allows objects to register themselves as observers of other objects. The observed object directly notifies its observers when there are changes to one of its properties [App06]. Note that these properties once more have to be key-value observing compliant and hence should follow certain naming conventions. Also certain required methods should be implemented in the observing and observed class.

With Cocoa Bindings, the key-value coding and observing mechanism is installed automatically when certain conventions are followed. Using the Interface Builder the programmer can visually specify all bindings present in the user interface. However, as it is advised to use controller objects as mediators, these have to implement the necessary

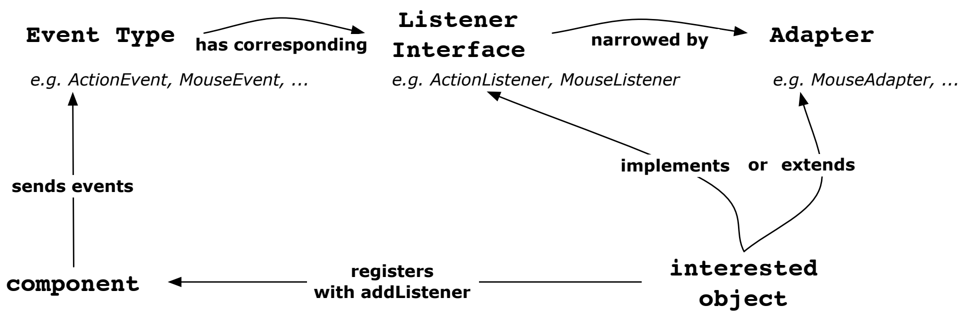


Figure 2.5: Event-based system in Java

UI behaviour. Also making decisions depending on context is to be implemented in the controller. *Hence the controller object provides for the connection between application and UI and now contains the tangled code.*

2.2.3 User Interfaces with the Java Application Builder

Several application builders exist for building applications and user interfaces in Java, such as Window Builder for Eclipse [Insc] and Matisse (a.k.a. Swing Builder) for NetBeans [Gul]. Although these application builders vary in detail, they all offer a similar approach for creating user interface applications.

While the behaviour of the application is implemented directly in Java, user interfaces can be created visually by means of tools. These tools offer a set of widgets that can be used in the interface, as well as the possibility to create custom widgets. These include text-fields, labels and buttons as well as layout managers.

Swing MVC

With Swing a modified MVC Structure was introduced into Java. Swing components have a separable model-and-view design where the model represents data of the application, and the view and controller as known in MVC are collapsed into a single UI object [Fow]. Both the event model and the data bindings mechanism described below, are used to update the view component if the model component has changed and vice versa.

Event Handling

The user interface is linked to the underlying application through Java's event handlers in both the Window Builder as in the Swing Builder. This mechanism is depicted in Figure 2.5. UI components generate events, such as action, mouse pressed, focus gained, component moved. Interested objects can register as a Listener for certain events with the component. When such an event occurs, the interested object will be notified. It has to implement an event handler which is the corresponding listener method that contains the code to handle the event [Zuk97, Gul, Insb]. For example in the code snippet below, the `TemperatureDialog` is a Listener (line 1) that registers itself with the `okButton` for action events (line 5). Upon such an event, the `TemperatureDialog` will be notified and the `actionPerformed` method (line 7–10) will be executed.

```
1  public class TemperatureDialog extends JDialog implements ActionListener{

2  private JButton okButton;
3  private void initComponents() {
4  okButton = new JButton("Ok");
5  okButton.addActionListener(this); }

6  private void close() { ... }

7  public void actionPerformed(ActionEvent e) {
8  if (e.getSource() == okButton) {
9  returnStatus = true;
10 close(); }}}
```

The event handling code provides the link between UI and application. If events have an effect on the application, it is the handler code that has to make sure the necessary behaviour is called. Also the UI changes that would result from an event being thrown, are to be implemented here. Note that this means that the *event handler will still contain tangled code.*

In Application builders, often event handler stubs are generated by tools which assign event handlers to components. The programmer is left with providing the necessary handling code, namely the actual business logic that should be triggered by the event. Furthermore these tools should be used with care, as the manually provided event handling code is lost upon re-generating the stubs, for instance when changing the UI later on.

Data Binding

On top of the event-based system Window Builder provides data bindings (a.k.a. value models) to link object properties with UI component values [Insa]. In essence this data binding mechanism uses the observer design pattern to inform components about changes in the object and vice versa. The Window Builder automatically generates the necessary code to install this observer behaviour. In this generated code, values used in the UI have a one-to-one relationship with object properties. If there is no one-to-one mapping and additional calculations are required for a property, it is the responsibility of the getter and setter methods of the property to take care of this.

2.2.4 User Interfaces with the Smalltalk Application Builder

The Visualworks Smalltalk environment was among the first to implement the model-view-controller pattern and still uses it today to separate the user interface from applications [Kra88]. Creating the connection between interface and application is accomplished by using an intermediate object (called application model) and value models.

The application model intermediate handles communication between the user interface and the domain model. Instead of accessing the domain model directly, UI components use the application model as their model. Whereas the domain model provides the system functionality, the application model provides the user interface processing capabilities [Hun97]. It thus contains the state of the user interface as well as part of the user interface behaviour [Fow06]. The application model can be considered as a specific interpretation of the domain model for a certain user interface. Linking the application model with the domain model such that the one reflects changes in the other, happens through the observer mechanism with the application model being an observer of the domain model.

Data Binding: Value Models

The change propagation mechanism between the application model and the UI components is handled by value models [Woo95] (see Figure 2.6). These automatically provide the update-change mechanism of the observer design pattern [Gam95].

The value models act as mediators to buffer the application model from the actual object it maintains. It wraps the object to be monitored and enforces a dependency on any changes of this value caused by the application [Tom00]. The application model no longer makes a direct reference to the actual object but instead refers to the value model. Accessing and updating the value of the object happens through the value model and when the actual object changes, it is the responsibility of the value model to inform its associated views that its value has changed and the view should display this new value. If the controller receives input which directly changes the value of the value model, the controller merely informs the value model that its value should change. This means the application model no longer needs to send change messages to its dependents and therefore no longer is concerned with any aspect of the view and controller at all [Hun97].

Visualworks Smalltalk provides simple value models, called Value Holders, to wrap application model objects. For example in the following the method `getCelciusTemperature` creates a `valueHolder` on a String object (line 3).

```
1  getCelciusTemperature
2     ^celciusTemperature isNil
3        ifTrue: [celciusTemperature := String new asValue]
4        ifFalse: [celciusTemperature]
```

For wrapping domain model objects, aspect adaptors are provided. The latter are value models that allow to specify what instance variable of an object is to be monitored. Upon changing this variable, the dependents of the adaptor are notified of the change.

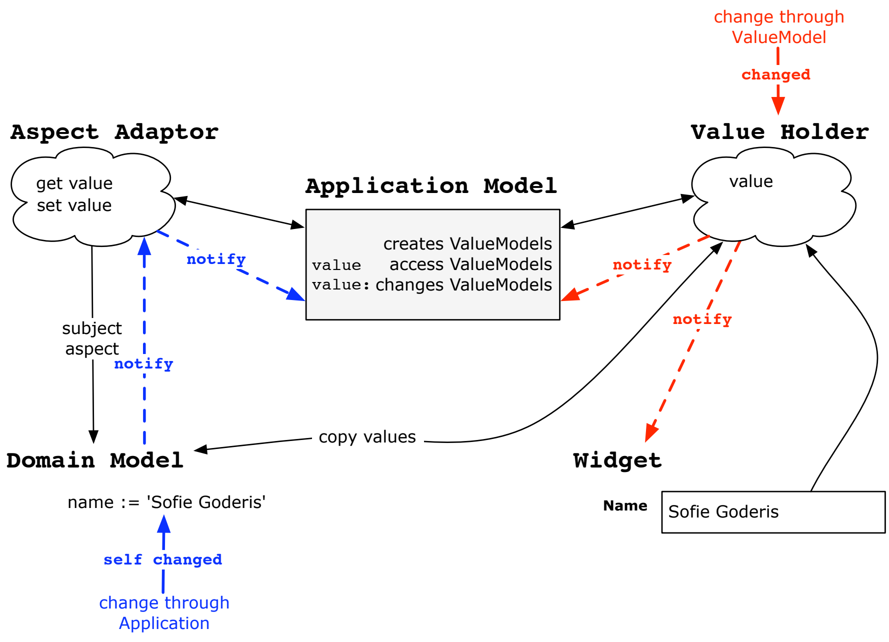


Figure 2.6: Value models in Visualworks Smalltalk

However, it is also common for a domain model to change its data independently of the application model, in which case the aspect adaptor is not notified of the change. If the application model depends upon these domain values however, it should be notified nonetheless. *In this case the programmer is responsible for having added the right '`self changed:#argument`' message to the domain model methods*[Vis02, Sha97]. *These messages will be scattered throughout the domain model, i.e. the application.* Consider the following example:

```
Method in TemperatureConverterApplicationModel class
celciusTemperature
   | adaptor |
      adaptor := AspectAdaptor subject: self temperatureConverter.
      adaptor accessWith: #celcius assignWith: #celcius:.
      adaptor subjectSendsUpdates: true.
   ^adaptor

Method in TemperatureConverter class
celcius: newValue
   celcius := newValue.
   self changed: #celcius
```

The user interface `TemperatureConverterApplicationModel` has an aspectAdaptor on the `temperatureConverter` application object (line 4), more specifically on its `celcius` variable (line 5). As the `TemperatureConverter` object needs to send updates upon programmatic changes, both the adaptor has to be informed (line 6) and the `TemperatureConverter` needs to notify its dependents by sending a self changed message (line 11).

Event Handling

Smalltalk events provide an additional approach to define dependencies between application and domain models such that both are kept coordinated. Events configure interactions between the UI and the application and domain models, and responses can be evoked on specified occasions. For example, using the standard dependency mechanism of Visualworks Smalltalk, an `ActionButton` widget sends its action message when the button is clicked, but using events the widget can evoke a response when it is clicked, pressed, tabbed into or out of, getting or losing focus, or when its label is changing [Vis02].

Methods to be called upon an event, are implemented in the application model (i.e. the intermediary between UI and application). Hence the actual link between UI and application is moved into this class. Note that this does not solve the tangling of application and UI code, but merely moved the problem of tangling to the application model class. *For instance, the code that deals with context changes as well as the code that is responsible for dynamic UI changes is dealt with by the models in the application model class and remains tangled.*

2.3 Other Approaches for Separating Concerns in User Interfaces

Although MVC is a widespread approach towards separation of concerns in user interfaces, other approaches have contributed in the field as well. Multi-tiered architectures have tackled a separation at the level of the architecture and model-based UIs at the modelling level.

Aspect-oriented programming does not consider the UI as a separate concern yet, but it is nevertheless an important technology when dealing with cross-cutting concerns. Another important part of the related work with respect to user interfaces, uses a declarative approach to tackle (part of) the UI specification.

2.3.1 Multi-tier Architectures

Multi-tiered architectures are often applied in software systems where communication with the user is important. They rely on the principle of modularity for achieving flexibility and scalability. Each tier, or module, is a functionally separated hardware and/or software component that performs a specific function [SUN00]. This modularity assures that each tier can be managed or scaled independently and therefore increases flexibility. As an example consider the modules in a 3-tier architecture which are [Mei00]

- user-services tier: front-end client that provides for communication with the user through some graphical user interface.
- business-services tier: business logic and interfacing between the user-services and data-services tier.
- data-services tier: back-end database server to store the business data.

Although multi-tier systems provide a separation of concerns with respect to user interfaces, they address this problem at a different level than MVC. MVC provides a pattern to separate the GUI from the application at the implementation level. Multi-tier applications provide an architecture to separate the GUI from the application at a conceptual level. It focusses on how to separate the user's view from the system's data by providing a middle-tier to act as a mediator between the two. This middle-tier contains the business logic and is responsible for handling the communication between the user and the data tier. The communication towards user or towards database is handled by middleware that allows for different communication patterns (e.g. synchronous, asynchronous communication, transaction management, ...). *How the programmer deals with separating UI logic and system logic at the middle-tier level is up to him*, for instance by using an object system. In multi-tier architectures the UI level is not supposed to know about the data level and the other way around. Note however that this shifts responsibility towards the middle-tier.

2.3.2 Model-Based User Interfaces

Model-based user interface development (MB-UID) is a paradigm for constructing interfaces [Suk94, Sze95] where rich user interface representations provide assistance in the user interface development process [Sze96]. The key idea is to represent all information about UI characteristics that are required for the UI development, explicitly in declarative models. Hence, by representing the different parts of UI by different models, MB-UID provides some form of separation of concerns for UIs. The runtime system executes these models in order to generate the working UI [Sch97]. At first MB-UID did not find wide acceptance, but as device independence emerged with context-sensitive and ambient intelligent environments, it revived [Mye00].

Types of Interface Models

Model-based UI development environments (MB-UIDE) provide an environment to construct and relate declarative models as part of the UI design process. Several declarative models have been distinguished [Sch96, Pin00a] and as MB-UIDE became popular within the context of context-sensitive systems some of these models were revised. The following models are used by several MB-UIDE approaches [Sch96, Pin00a, VdB04, VdB05, Lu07]. Note that not all models are necessarily supported by a particular approach.

Application Model Some MB-UIDEs include an application model which is very similar to a domain model [VdB04] as defined in contemporary software engineering methods. This model describes the properties of the application that are relevant to the UI [Pin00a]. In certain approaches the application model is actually a standard domain model with a basic extension to capture user interface specific information [Sch96].

Task-Dialogue model The Task-dialogue model, also known as the activity or the interaction model, combines the task and the dialogue model [VdB05]. The task model describes the tasks end users plan to carry out with the developed interactive system. It shows a hierarchical view on the activities that need to be accomplished using the modelled interface. The task model is sometimes combined with an interaction model which is used to model the possible interactions between user and interface [Sam04]. The dialogue model concentrates on how these activities are organised in the user interface and on what tasks are available at what time. It describes the human-computer conversation which consists of invoking commands and specifying inputs by the end-user and querying and presenting information to the end-user [Pin00a, VdB04].

Presentation Model The presentation model describes what the UI looks like for the end-user, such as the components on the display, their layout characteristics, and the dependencies amongst them [Sch96]. Some MB-UIDEs split the presentation model into an abstract an concrete model. The abstract presentation model provides a conceptual description of the structure and behaviour of the visual parts of the user interface. Here the UI is described in terms of abstract objects. In the concrete presentation model they are described in terms of widgets and this model describes in details the visual parts of the UI [Pin00a]. The concrete presentation model will be used to describe the user interface for a specific set of contexts or platforms. Remark that the dynamic part, which shows application-dependent data and typically changes during runtime, needs to be *programmed using a general purpose programming language and a low-level graphical library* [Sch96].

User Model The user model models describes the characteristics of the end users of the system being developed. This can be individual users or groups of users. These models allow to create individual (personalised) user interfaces adapted to the individuals needs [Sch96]. The user model captures capabilities and limitations of the user population, for instance the kind of interaction techniques that are available for visually handicapped people differs from the the techniques for other.

Context Model As context-sensitive systems became more prominent, also the context model became an important part of MB-UIDE. Although the user model can also be considered to be part of this model [Sot05], context is also influenced by environment and platform. The context model contains the concepts that directly or indirectly influence the interaction of the user with the application. Also the relation between the concepts is important as well as how information is gathered [VdB05].

Generating the Runtime Application

Most MB-UIDEs are used in combination with existing software environments to generate executable programs from declarative models [Pin00b, Cle06]. The declarative UI models are transformed into generated source code by using techniques similar to model-driven engineering techniques [PM07]. As such, model-based UIs and model-driven engineering are merging. This implies that the developers also provide the transformations to map high-level models onto lower-level models onto application code [Bot06, Sta08].

With the rise of context-sensitive systems, the need has become apparent to be able to access the declarative models (or the information contained within these models) at runtime. Context-sensitive user interfaces adapt to the context and need to reflect (dynamic) changes in context into the user interface when appropriate [Sot05, VdB05, Loh06]. Also for 'plastic' user interfaces [Sot05] where a UI has to withstand variations of context (user, platform, environment), models should live at runtime such that all abstraction levels and traceability links are available during execution in order to compute adaptation [Dem06, Sot08]. This 'second' generation of MB-UIDE also uses a model-driven engineering approach to map model transformations onto the runtime application dynamically.

2.3.3 Aspect-Oriented Programming Techniques

Aspect-Oriented Programming (AOP) achieves a separation of concerns by modularising crosscutting concerns into aspects. Each concern is expressed by its own aspect or set of aspects. An aspect weaver will instrument the base program with the crosscutting concerns in order to obtain the final application. To our knowledge, AOP has not yet been considered in the context of the UI as a concern, although AOP has been applied to part of the UI by implementing certain design patterns with aspects.

In [Paw05] AOP is applied for J2EE applications. Such an application follows a tiered architecture and uses a business tier to form the interface between the client tier and the enterprise (data) resources. Several best practices and patterns in J2EE are applied for both the business and the client tier, such as the service locator pattern, the session facade, the business object pattern, etc. Most of these patterns can be implemented using AOP and by doing so eliminate the crosscutting code that is otherwise required to achieve the pattern's implementation. The application's Java Swing user interface is implemented in the client tier and its web representation in the presentation tier. Aspects are applied to introduce communication, input verification and localisation patterns. Although these aspects tackle UI related issues, they focus on low-level implementation issues and do not consider a full separation of the UI concerns from the underlying application. *The code responsible for linking the interface with the application remains tangled. Adapting for highly dynamic context-sensitive systems for instance would still require ad-hoc mechanisms to be implemented.*

Several design patterns [Gam95] benefit from an aspect-oriented implementation [Han02, Mil04]. As discussed above, the controller component in MVC uses a notification mechanism to link the model to the views. Using AOP to implement the observer design pattern localises the dependencies between the different participants in the pattern code (i.e. the aspect). As such, a dependent becomes oblivious to its role in the mechanism and can be reused in different contexts without modifications or redundant code. This ‘automatic’ installation of the pattern solves the developers’ problem with manually maintaining the pattern code, such as adding the necessary notify-change messages. The aspect version of the observer pattern makes models truly oblivious to their views.

Nevertheless current implementations of the observer pattern suffer from the fragile pointcut problem, which means that changing or evolving the base program requires the observer aspects to be revised. Additionally, *using the observer pattern does not bring salvation for handling the complex control flow structures that deal with dynamically adaptable user interfaces.*

Aspects have been used to gather information about the UI and application dynamics, such that this information can be used in a graphical editor to give the programmer a dynamic view on where widget properties are adapted [Li08]. With this approach, the UI code remains in the same location, entangled with the application, but the editor makes it easier for the programmer to focus on it. Although this approach does not create a separation of user interface concerns, it does consider the user interface to be an important and cross-cutting concern.

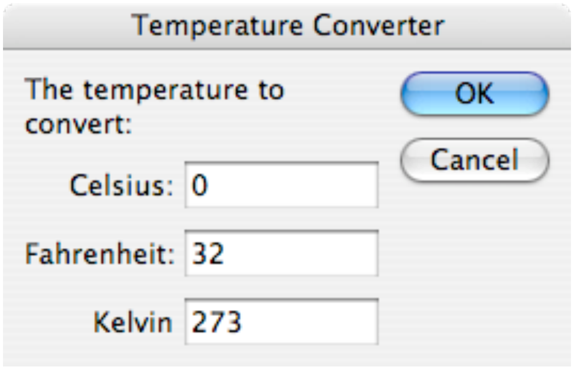
2.3.4 Declarative Approaches and Techniques

Quite some approaches have implemented part of the user interface by using a declarative approach. For instance Adobe uses declarative declarations for laying out the view and for the data bindings between UI components and their value. JGoodies Forms provide declarations for UI layout. Other approaches split the user interface into several components in analogy with MVC and model-based UIs and use a descriptive language to describe these components. XUL and UIML are both XML compliant languages that provide such declarative descriptions. Other variations on these approaches do exist, but in what follows we will discuss the above mentioned examples.

Adobe Adam and Eve

The Adobe Source Libraries (ASL) are a set of commercial libraries developed by Adobe to construct applications [Par07]. The two significant libraries in ASL are the property model library (Adam) and the layout library (Eve). They aid in modelling the user interface appearance and behaviour.

Adobe applications typically use a traditional model-view-controller pattern. The model represents the document being edited. The view is the display of the document in a window. The controller is a collection of commands that can be used to modify the document. Adam and Eve are used for specifying the view and controller part. A property model in Adam contains the logic to handle events, validate input, setting up and tearing down components. Hence, Adam provides the controller part of interface components. Eve is used to construct the user interface where components are laid out



Celsius: 0
Fahrenheit: 32
Kelvin 273

Figure 2.7: A user interface created with Adobe source libraries

on the screen and linked to certain actions. Therefore, Eve provides the view. The resulting user interface component combining both, is independent from the model and allows the designer greater flexibility in changing the layout and the choice of interface elements without impacting the underlying model [Par07]. Both Adam and Eve use declarative descriptions. Each has a dedicated parser to transform the descriptions into workable code to interact with the application (model) code.

The interface in Figure 2.7, adopted from [Par07], gives a feeling of Adam and Eve's declarative nature.

An Adam declaration consists of a 'sheet' (i.e. structure) containing several 'cells'. It is used to define the interface's data bindings as follows:

```
1  sheet temperature_converter {
2     interface:
3        fahrenheit   : 0;
4        celsius      : 0;

5     logic:
6        relate {
7           fahrenheit <== scale(x: celsius, m: 9/5, b: 32);
8           celsius    <== scale(x: fahrenheit, m: 5/9, b: -32*5/9);}

9     output:
10       result <==  {t1: celsius, t2: fahrenheit};}
```

The interface has two data elements, namely `fahrenheit` and `celsius` (lines 2–4). These are related with one another through the `relate` statement (line 6), such that changing one affects the other by applying the temperature conversion. The output of the interface, which is transmitted to the underlying model upon clicking the `ok` button, is specified on line 9. That this output should be transmitted upon clicking the `ok` button is specified with Eve, together with other view aspects. The declarations for the example are:

```
1  layout temperature_converter {
2     view dialog(name: "Temperature Converter")

3     { row(child_horizontal: align_fill)

4       { column(child_horizontal: align_fill)
5         { static_text(name: "The temperature to convert:",characters:10);
6           edit_number(name: "Celsius:", bind: @celsius);
7           edit_number(name: "Fahrenheit:", bind: @fahrenheit);
8           edit_number(name: "Kelvin", bind: @celsius, scale:[1, 273.15]);}

9         column(child_horizontal: align_fill)
10        { button(name: "OK", action: @ok, default: true, bind: @result);
11          button( name: "Cancel", action: @cancel);}}}}
```

Buttons and components are positioned in rows and columns on lines 3, 4 and 9. The `ok` button as specified on line 10 has a name and a default value. Its action specifies what happens if the button is clicked, namely the `ok` event is triggered. These events are predefined in Eve. The `bind` argument links the button and its action to the value of one of the interface elements (as defined in Adam). In this example, clicking the button is bound to the value of `result`, which was specified in Adam as being both the `celsius` and `fahrenheit` temperature. `edit_number` is a predefined 'widget' that consists of a label and an input field for numeric values. The widget on line 6 uses Celsius as a label and will bind the value of the inputfield to the `@celsius` variable (used in Adam's interface declaration). The widget on line 8 binds the value of the input field of the Kelvin temperature to the value of the `celsius` variable after it has been filtered through the scale filter, i.e. converted from Celsius to Kelvin. This filter works both ways. If for instance the Kelvin input field is changed, the filter will adapt the value of the `celsius` variable as well. *Note that this implies that the logic for converting the temperature can be written at two different places, namely either as part of Eve or Adam. This ambiguity can complicate the developers' comprehension of the UI and application interactions.*

As these examples show, Eve is used to specify the layout of a user interface and the bindings between the user interface components and values of the property model. Adam specifies the property model by means of parameters (which can be transferred to the application). These parameter fields can depend upon each other. These dependencies are specified with Adam declaratively. As values change, dependencies are updated. Note that in essence Adam and Eve provide a data binding mechanism to fulfill the controller part of the MVC model as was discussed in Section 2.1.

JGoodies Forms

The JGoodies Swing Suite provides components and solutions that complement Swing to solve common user interface tasks [jgo]. One of the libraries, called JGoodies Forms, deals with the layout of the user interface and uses declarative specifications to do so. JGoodies focusses on form-oriented panels but can be used for the vast majority of rectangular layouts [Len04] as well. JGoodies Forms wants to deal with the fact that often layout systems are difficult to understand, are hard to work with, and do not represent the human mental layout model. In addition their implementation does not separate concerns such that the general layout is mixed with the actual components of the UI and their properties. This makes it impossible to reuse layouts. Therefore JGoodies separates the layout specification. Moreover it uses an expressive language to write these specifications supposedly in close relation to the human mental model.

Components are aligned vertically and horizontally in a rectangular grid of cells, which each component occupying one or more cells. To define a form layout with JGoodies one specifies the forms columns, rows and optionally column groups and row groups. For example, the temperature converter UI example of Figure 2.3 can be specified with JGoodies as follows:

```
1  FormLayout layout = new FormLayout(
2     pref, 4dlu, 50dlu, 4dlu, min, // columns
3     pref, 2dlu, pref, 2dlu, pref, 2dlu, pref); // rows

4  layout.setRowGroups(new int{{1, 3, 5, 7}});

5  JPanel panel = new JPanel(layout);

6  CellConstraints cc = new CellConstraints();

7  panel.add(new JLabel(Celcius), cc.xy (1, 1));
8  panel.add(celciusTextField, cc.xyw(3, 1, 3));
9  panel.add(new JLabel(Fahrenheit), cc.xy (1, 3));
10 panel.add(fahrenheitTextField, cc.xyw (3, 3, 3));
11 panel.add(new JLabel(Kelvin), cc.xy (1, 5));
12 panel.add(kelvingTextField, cc.xyw(3, 5, 3));
13 panel.add(okButton, cc.xy (1, 7));
14 panel.add(cancelButton, cc.xy (3, 7));
15 panel.add(resetButton, cc.xy (5, 7));
```

Lines 1–3 construct a new FormLayout and specify columns and rows using a string representation, with pref being the preferred size, min begin the minimum size, and dlu a size unit that scales with the font. Line 4 indicates that all component rows shall get the same height. Next, a layout container (i.e. JPanel) is created (line 5) and component locations are specified in the grid (line 6) through the CellConstraints object. Finally, the panel is filled with the components on lines 7–15 [Len04]. Note that the

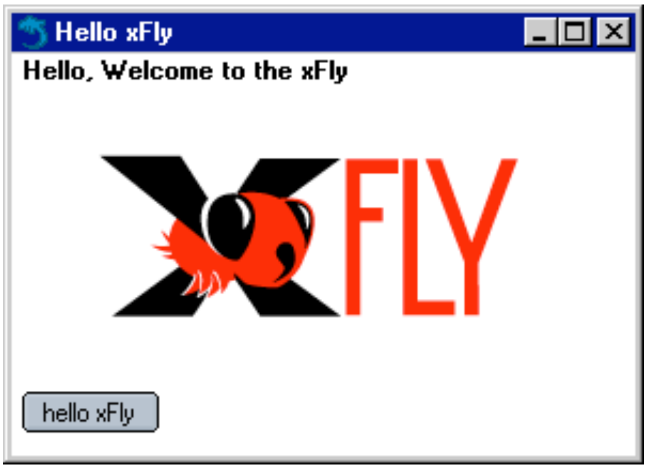


Figure 2.8: A user interface created with XUL

declarative statements in JGoodies aid developers in thinking about the layout of the UI in a more natural way. Nevertheless the *declarative statements should be sufficiently expressive* and abstract such that describing the declarative layout does not become overly complex.

User Interface Description Languages

A User Interface Description Language (UIDL) describes the characteristics of interest of a user interface with respect to the rest of an interactive application [Sou03]. It often consists of a declarative language for capturing the essence of what a UI is or should be, independently of physical characteristics [Van04]. Similar to model-based UIs, UIDLs distinguish between several models (or concerns) in a UI. Often an XML-compliant language is used to express these models. As several UIDLs exist we limit ourselves here to XUL and UIML. XUL became quite popular through its use in Mozilla. UIML is becoming a standard when expressing context-sensitive user interfaces. Note that also UsiXML [Van04] provides a similar standard.

XUL The XML User Interface Language (XUL) is a markup language for describing user interfaces that was developed by the Mozilla project for use in its cross-platform applications. Mozilla is used by developers for creating applications that can be installed locally or run remotely over the Internet [Moz07a]. Separate UI descriptions are used to assure that the arrangement and even presence or absence of controls in the main window is not hardwired into the application [Moz07b]. Hence XUL provides a separation of UI concerns that are related to its visualisation.

A XUL interface typically consists of [Dea07]

- *Content*: the declarations of windows and user interface elements contained within them. These elements will define the layout of the user interface.
- *Skin*: The style sheets and image files, which describe the details of the appearance of a window.
- *Locale*: the text that is displayed within a window. This allows for software localisation with for instance text in a particular language.

The example in Figure 2.8 shows a small user interface created with XUL (adopted from [Moz07a]). It has the following XUL contents file:

```
 1  <window title="Hello xFly"
 2    width="300"
 3    height="215"
 4    onload="centerWindowOnScreen( )">

 5  <vbox align="left" id="vb">
 6    <label id="xlabel"
 7      value="Hello, Welcome to the xFly"/>
 8    <image src="http://books.mozdev.org/xfly.gif"/>
 9    <button label="hello xFly" oncommand="alert('hello.');"/>
10  </vbox>
11  </window>
```

Several XUL interface elements are specified in this example together with some attributes. For instance the element on line 1 is a window with a title, width, height and onload attribute. The XUL element on lines 5–10 is a vertical layout box (`vbox`) with three elements, namely a label, an image and a button. XUL provides vertical and horizontal layout boxes, which can be combined into grids, to specify the layout of the interface elements. In addition alignment statements can be used as shown on line 5 which will left align all the components in this particular vertical layout box. The `oncommand` attribute on line 9 specifies what event handler to call when the button is clicked. The event handler in this example is one of the generalised event listeners provided by XUL. However, XUL can make use of javascript to provide code to deal with events. For example, the button could be specified as

```
<button label="hello xFly" oncommand="greet( );"/>
```

where the greet command is specified in a javascript file as

```
function greet( ) alert("hello");
```

In addition, the XUL interface can use a style file to separate the actual presentation of XUL elements from its contents. For instance, the following is a style for the `xfly` interface:

```
#xlabel  font-weight: bold;
window   background-color: white;
```

Finally, in order to provide different localisation for a XUL interface, the parts that are written in a particular language are to be generalised. For instance, the following localisation file contains the labels for the XUL interface in English:

```
<!ENTITY label.val       "Hello, Welcome to the xFly ">
<!ENTITY btn.lbl         "hello xFly ">
```

The XUL elements in the contents file are replaced with :

```
<label id="xlabel" value="&label.val;"/>
<button label="&btn.lbl;" oncommand="greet( );"/>
```

In order to use javascript, style files and localisation files, additional statements have to be added to the XUL contents file in order to incorporate the necessary files. The complete XUL contents file is as follows

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin" type="text/css"?>
<?xml-stylesheet href="chrome://xfly/skin" type="text/css"?>
<!DOCTYPE window SYSTEM "chrome://xfly/locale/xfly.dtd" >
 <window title="Hello xFly"
   xmlns:html="http://www.w3.org/1999/xhtml"
   xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
   width="300"
   height="215"
   onload="centerWindowOnScreen( )">
 <script type="application/x-javascript"
   src="chrome://global/content/dialogOverlay.js"/>
 <script type="application/x-javascript"
   src="chrome://xfly/content/xfly.js"/>
 <vbox align="left" id="vb">
   <label id="xlabel"
     value="&label.val;"/>
   <image src="http://books.mozdev.org/xfly.gif"/>
   <button label="&btn.lbl;" oncommand="greet( );"/>
 </vbox>
</window>
```

Style sheets, scripts and localisation are used to separate the XUL contents (UI elements) from their actual presentation and behaviour. These can be changed or replaced with other versions without affecting the contents file. *The alternative representations have to be provided beforehand and cannot be updated dynamically.* Furthermore, the contents

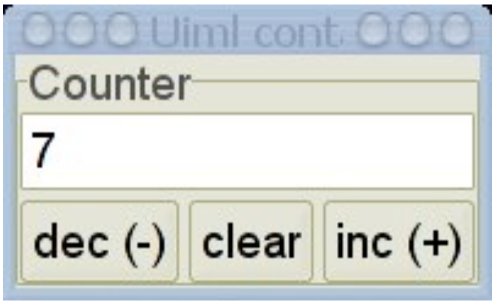


Figure 2.9: A user interface created with UIML

file needs to be aware of what style, scripts and localisation files to use. *The statements to link to these other files are spread throughout the contents file* [Moz07b]. This makes XUL interfaces little customisable, for instance when wanting to apply UI guidelines for a certain company to all of the interfaces used in that company. Note that XUL interfaces can trigger an underlying application through javascript, but *the opposite link from application to UI is absent.*

UIML The User Interface Markup Language (UIML) is an XML-compliant language designed to build interfaces that can be deployed on multiple appliances [Abr99]. UIML is used to describe the appearance of a UI (e.g. widgets, colours, fonts, layout), the user interaction with the UI (e.g. what happens if a button is clicked), and the connection between UI and application. UIML is used to specify the interface which will be rendered by possibly different renderers. UIML is particularly well suited for generating UIs that are deployed to different devices.

The following is UIML code for the interface shown in Figure 2.9.

```
<uiml>
 <interface>
  <structure>
   <part class="Frame" id="CounterFrame">
   <part class="VBox">
    <part class="Entry" id="counterEntry"/>
     <part class="HBox">
      <part class="Button" id="dec"/>
      <part class="Button" id="clear"/>
      <part class="Button" id="inc"/>
     </part>
    </part>
   </part>
  </structure>
  <style>
   <property part-name="CounterFrame" name="label">Counter</property>
   <property part-class="Button" name="font">Times</property>
   <property part-name="clear" name="label">clear</property>
   <property part-name="dec" name="label">
    <reference constant-name="dec"/></property>
    <property part-name="inc" name="label">
     <reference constant-name="inc"/> </property>
    <property part-name="counterEntry" name="text">0</property>
  </style>

  <content id="English">
   <constant id="dec" value="dec (-)"/>
   <constant id="inc" value="inc (+)"/>
  </content>
  <content id="Dutch">
   <constant id="dec" value="min(-)"/>
   <constant id="inc" value="plus(+)"/>
  </content>

  <behavior>
   <rule>
    <condition>
     <event class="ButtonPressed" part-name="inc"/>
    </condition>
    <action>
     <property part-name="counterEntry" name="text">
      <call name="MyMath.inc">
       <param><property part-name="counterEntry" name="text"/></param>
      </call>
     </property>
    </action>
   </rule>
   ...
  </behavior>
 </interface>

 <peers>
  <presentation base="http://research.edm.luc.ac.be/kris/projects/
                           uiml.net/gtk-sharp-1.0.uiml"/>
  <logic id="mmath">
   <d-component id="MyMath" maps-to="MyMath">
    <d-method id="inc" return-type="string" maps-to="Increment">
     <d-param type="System.String"/>
    </d-method>
    <d-method id="dec" return-type="string" maps-to="Decrement">
     <d-param type="System.String"/>
```

```
59        </d-method>
60       </d-component>
61     </logic>
62   </peers>
63 </uiml>
```

The interface part (lines 3-14) has a structure, style, content and behaviour. The structure specifies which elements are to be shown in the interface and how they are organised. One interface can have several of these structures specified, for instance for a desktop PC and a voice interface. The different elements of the UI are listed and given an identifier.

The style (lines 15-24) transforms the high-level XML statements into lower-level interface dependent components. For example on line 18, the button with name (id) `clear` is given a label 'clear'. Line 17 declares all widgets of the class 'Button' to be in a certain font. Instead of specifying a property here, it can be linked with the contents part of the UI. For instance, the label of the decrement button is a reference to a content element with name 'dec' (line 19). This element is specified in the content part of the interface (lines 25–32). This part will contain application-dependent data, which is text for each content element. What content to apply is specified by passing its `id` as an argument when rendering the UI.

The behaviour part (lines 33–47) describes the actions that occur when a user interacts with the interface. It is built upon rule-based languages and each rule has a condition and a sequence of actions. The actions can change a property of some part in the UI, invoke a function or method in a scripting language or from a backend object, or throw an event [Abr04]. *Unfortunately all changes have to be anticipated in advance (at design time), whereas in current context-sensitive systems it is desirable to be able to cope with dynamic UI changes. In order to do so, the rule-based mechanism lacks the expressiveness and inferencing capabilities of a full declarative programming language.*

In addition to the interface part, a peers part (lines 49–62) can be added to map property and events names used elsewhere in the UIML document to application logic. Hence, it provides the link from UI to the backend application. Additionally it is possible for the application to subscribe to certain events from the UIML interface. *Currently UIML does not support events being triggered from within the application logic, which would allow the UI to respond to application changes.*

UsiXML Similar to UIXML, UsiXML describes UIs for multiple contexts of use and by doing so supports multi-modal user interfaces in which interaction techniques, modalities of use, and computing platforms can vary. These variations are described at an abstract level such that user interface design becomes independent of characteristics of physical computing platform [sit]. UsiXML uses a set of basic UI models to deal with the development of multi-directional UIs, namely the AUI model, CUI model, domain model, task model, context model, mapping model, and a transformation model.

The AUI model specifies the abstract user interface by defining abstract interaction objects for each concept and interactions between several of these objects. The CUI specifies the concrete user interface by concretising the AUI for a given context into concrete interaction objects. It makes the look and feel of a UI explicit and is specific for a particular environment.

The task model describes the interactive tasks as they are viewed by the end user interacting with the system. The domain model is a description of the classes of objects that are manipulated by a user while interaction with a system. The mapping model contains a set of inter-model relationships. The context model contains a user model, a platform model and an environment model. These describe the three aspects of use in which an end-user is carrying out interactive tasks [Van04].

Several tools have been developed on top of UsiXML in order to aid the programmer in specifying the different models. Third-party rendering engines are used to transform the UsiXML interface into a running application.

2.4 Conclusion

The model-view-controller metaphor, although its interpretation changed over the years, is still used in software engineering practices to separate the user interface from the underlying application. The underlying application is represented by the model, the user interface by the view (comprising both output and input) and the controller acts as a mediator between the two to link them together. Linking the two together is done through a notification mechanism. This can be either through events, through a data binding mechanism, or both.

Other approaches such as Adobe's Adam and Eve provide a declarative medium for specifying events and data bindings. JGoodies focusses on a declarative layout specification, on top of Swing's MVC structure. XUL and UIML use an XML-like style to provide declarative UI specifications where also view and model are separated from another. Also UI is linked with the application this way, but no way has been provided to link the application back to the UI.

Although all of the above mentioned approaches tackle separation of concerns in user interfaces to some extent, we will discuss their shortcomings in Chapter 3 and introduce a set of requirements to achieve a true separation of concerns in user interfaces.

Chapter 3

A Foundation for Separating User Interface Concerns

Separating concerns is known to increase reusability and maintainability of a software system, not in the least because it increases the developers' comprehension of the system. The approaches discussed in Chapter 2 achieve separation of concerns to a certain extent, but as is discussed in Section 3.2 the separation is not fully attained and therefore the programmer is still confronted with difficulties when maintaining and evolving a software system. MVC attempts to support developers in implementing the user interface in separation of the application, but they are still confronted with the implementation details of the mechanisms to do so. Other approaches like model-based UIs put a lot of effort in modelling the UI but fall short in carrying it through to the implementation level. The approach we discuss in this dissertation will overcome these problems and achieve a full separation of user interface concerns at the implementation level as well.

Section 3.4 will provide the foundations for our approach by explaining the set of requirements which a solution should adhere to in order to provide the desired separation of concerns. However, first a common vocabulary for the different concerns encountered in user interfaces will be introduced in Section 3.3. This vocabulary is needed as various UI approaches use similar terms but not always attribute the same meaning to these terms. Furthermore most approaches pay little attention to the link between UI and application, although this is the main reason for tangled and scattered code when it comes to user interfaces.

Finally, in Section 3.6 we conclude by a high-level methodology to be followed when using a solution that supports a separation of user interface concerns.

3.1 A Calculator Application as Running Example

Before we zoom in on the separation of user interface concerns, we introduce an example that is used as an illustration throughout the remainder of this dissertation. This example is a calculator application as shown in Figure 3.1. The standard version of the calculator (Figure 3.1a) has buttons for numerical input and for performing basic calculations. These include addition, subtraction, division and multiply, as well as changing sign and resetting the calculator. This basic calculator behaviour is extended with the extra functionality that upon clicking the divide button in the basic calculator, the zero button gets disabled as divisions by zero are not allowed.

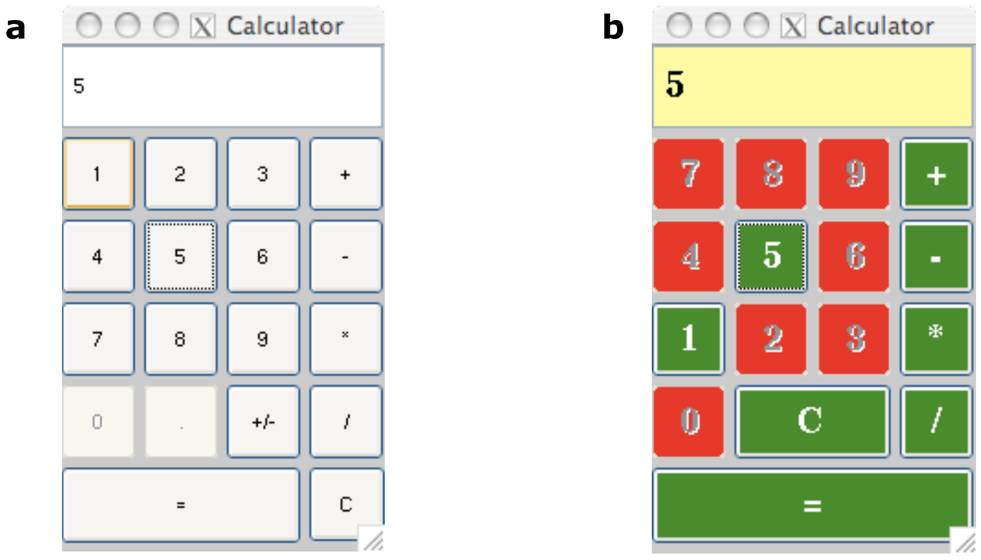


Figure 3.1: Two modes for a calculator: a) standard b) simplified

An alternative version of the calculator is a simplified version shown in Figure 3.1b. This calculator does not allow to change the sign of numbers and is restricted to work with whole numbers only. Therefore divisions that lead to decimal numbers are prohibited by disabling all digit buttons that result in a decimal if used as a second argument in the division (e.g. as 5 can be divided by 5 and 1, only these buttons are enabled). The calculator uses colourful buttons. Buttons that are not allowed are disabled and coloured red. Buttons that are allowed are enabled and coloured green. Note that both versions of the calculator can use the same underlying application logic. Therefore one can consider both calculator versions to be a different view (or UI) on the same application.

3.2 Analysis of Existing Work

All the approaches discussed in Chapter 2 separate the user interface from the application to a certain extent. Unfortunately this separation is not always fully achieved and as a consequence the programmer is left with a difficult task when evolving and maintaining either the user interface, the application, or its connection. Furthermore new challenges in software systems, like context-sensitive systems [Duc01] and agile development [HI00], reinforce the need for a clear cut separation of user interface concerns.

New Software Challenges *Ambient Intelligence* (AmI) [IST03] is a vision of ubiquitous computing where mobile devices are omnipresent in our every day environment. The devices cooperate and interact with each other and their environment. Applications are no longer isolated, but can migrate from one device to another. As one application will be used on different devices, the user interface should also adapt to the device it is running on for not all devices have the same characteristics and don't support the same in- and output means. In addition, the environmental properties in which the device is used, might change over time. These context changes require applications to adapt their behaviour accordingly in order to meet the user's expectations more closely. Hence, applications need to be context-aware [Des07]. This context-sensitivity also challenges the user interface implementation which needs to behave or present itself differently according to the current usage-context. Consider for instance a different presentation because of spatial information indicating whether the device is being held in landscape or portrait mode. Related work with respect to context-sensitivity focusses on a language engineering approach [Cos05] and not on UI aspects, although the impact of context-sensitivity on user interfaces is not negligible. In the context of the CoDaMos research project a set of key challenges in the area of Ambient Intelligence is investigated. Personal devices will form an extension of each user's environment, running mobile services adapted to the user and his context [cod05]. The fact that both Ambient Intelligent systems and context-aware systems put software under continuous strain to adapt to different user capabilities and changing usage contexts, imposes an even stronger need for separation of concerns in user interfaces.

Also in the context of *Agile development* [HI00], working software systems are continuously delivered and regularly adapted to changing circumstances. Late changes in requirements are also possible. This requires support for rapidly changing systems, also with respect to the time spent in making the changes. Obviously this imposes constraints on the user interface development as well, as these have to change along with the system. Recent work in concept-centric coding [Der06] acknowledges this problem and proposes a solution where domain knowledge is actively used in the development of software systems. By doing so it is extremely flexible for providing for software variability. Unfortunately until now it does not offer support for user interfaces, although UIs also contain quite some variation points with respect to one another [God05b].

Properties of an Application with respect to its User Interface Not all software systems have the same UI needs. For some systems it is sufficient to provide simple form-based UIs that provide a view on the underlying data model. Other systems require different views on the same application where views are also updated if changes did not happen through that view. And UIs of yet other systems are influenced by its business (domain) knowledge or its context-sensitivity, as for example when the UI needs to behave or present itself differently according to the current usage-context. This requires user interfaces to change dynamically.

In this dissertation we focus on highly flexible user interface code for object-oriented software systems where the following properties of the application should be kept in mind:

- *application and UI are linked in both ways such that changes made to the application are reflected in the UI, even if those changes were not inflicted through that UI.* For example the standard calculator example of Figure 3.1 shows the computed result on its display. If the computed result changes in the application because of an internal computation which was not initiated through the calculator UI, this calculator should nevertheless show the newly computed result.

- *different views on the same application can exist.* One application can provide alternative views on its internal data. For example a user's home address is shown textual in one view while it is displayed geographically on a map in a different view. Also when different users share an application for collaboration purposes, they will each use their own view on that same application. This view can be either an instance of the same view, or a different view for each user. For instance two users are inputting financial transaction statements by means of an instance of the same textual input interface, while a colleague is adding digital invoices to these transactions through an upload interface. Other applications run on a range of devices (e.g. pda and desktop computer) and will provide an adapted view for each of these particular devices.

- *dynamic UI changes are needed.* These changes include adding and removing components, repositioning components and changing properties of components as well as changing the navigational flow between the different windows of an interface. Additionally, when new components become available, new behaviour becomes applicable. Also the behaviour of existing components can change. For example, when switching from the standard calculator to the simplified version (e.g. through a menu choice), the behaviour of the divide button changes. Furthermore, upon clicking the divide button in the simplified calculator, the colour property of its buttons is possibly changed.

In addition we would like to stress that in this dissertation we focus on the actual implementation of a user interface and not on how it should be modelled, nor on how it lives up to a good user-centric design. Although these are interesting research topics, we focus on the lack of separation of concerns in the implementation of user interfaces, and especially of their interaction with the underlying application.

3.2.1 Linking UI and Application in Both Ways

The UI and the application can be linked in two directions. Firstly, a user interface can trigger behaviour in the application. For instance in the calculator example clicking the equals button triggers the application to compute the result for the expression that was entered. Hence, the user interface event of clicking a button is linked to the compute message call of the application. Calling a message in the application upon an event occurring in the user interface, is also referred to as a 'call-back'. These call-backs sometimes result in returning a value to the user interface such that the latter gets updated accordingly. Secondly, actions in the application that were not initiated by the user interface, can also have an effect on the user interface. These actions are for instance triggered by an internal application computation or by another view on the same application. At this point, the application needs to know about the user interface that is to be updated. Hence, the need for the (opposite) link between application and user interface becomes apparent.

Linking UI with Application

As discussed in Chapter 2, linking the UI to the application is often resolved by event handlers. The event handler is triggered upon an event in the user interface. The handler provides the code to call updates on the application and the user interface. This code is responsible for calling the necessary application behaviour as well as for providing the UI changes that results from the event being thrown. For example, clicking the divide button in the calculator is a user interface event upon which an event handler is called to deal with the event. The handler is responsible for letting the application know that the divide operator is chosen, but also to disable buttons that have become invalid to use. Hence the event handler contains code for both updating the application and updating the user interface, and therefore contains entangled code. Furthermore developers have to implement this event handling code and are confronted with the code tangling as well as with the low level mechanisms that are used to call application behaviour and to update the user interface. For instance for the calculator, the developers need to know how to change the colour of a user interface component. If for instance the user interface update would require to reposition components, this updating the UI becomes more complex as it often requires code to recalculate layout positions.

This communication flow from UI towards application where the UI makes calls towards the application, and possibly updates itself according to the result of that call, is used extensively in web-based applications. On the contrary these applications lack the mechanism where the application makes calls to the UI, as will be explained next.

Linking Application with UI

Whereas web-based applications and tiered architectures typically target applications with a link from user interface to application (call-backs), applications using an MVC-like mechanism do allow for the opposite link to occur as well. In MVC the notification mechanism to provide this link is often taken care of by the observer pattern. For instance this pattern is used in Java, .NET and Smalltalk Visualworks to link the application to the UI, such that UIs are notified of changes in the application. Upon these notifications, the UI is responsible for undertaking the necessary actions to reflect these changes.

The major issue when using the observer pattern is that the application code is contaminated with update statements. When application data is changed or an application method is triggered, the application is responsible for calling the updated/changed message that informs all its dependents of the change. These updated/changed statements usually have to be added by the programmer manually. When changing where and when notifications take place, the application code itself needs to be changed. This makes reusing and maintaining the application vulnerable to errors. Note that aspect-oriented development has been used to inject the notification code into the application [Han02]. Aspect-oriented languages using a logic-based crosscut language such as Carma [Gyb03], model-based pointcuts [Kel06] or generic aspects such as logicAJ [Rho06] and JAsCo [Van05], have shown how to capture these state changes in a generic way. As these languages de-couple aspects from the actual implementation, they do not suffer from the fragile pointcut problem as other aspect languages do. The fragile pointcut problem refers to the fact that in some crosscut languages, pointcut definitions rely heavily on the structure of the base program. This tight coupling of the pointcut definitions to the base programs structure and behaviour can hamper the evolvability of the software because it implies that all pointcuts of each aspect need to be checked and possibly revised whenever the base program evolves [Kel06].

3.2.2 Different Views on the Same Application

Some software systems require different views to be available for the same application. In collaborative systems for instance different users work on the same underlying application through their own view. This can be either their own instance of the same view (user interface) or a different view. For example one user works on a document through a English user interface while another user used an Arabic version of the interface with differently ordered and displayed components. Note that in this dissertation we do not deal with how collaborative applications are implemented. What we are interested in is the user interface code for such applications.

In other software systems, one user can use different views to interact with the same application. For instance statistical data is shown textual in one view and by means of a pie-chart in another. Also will the application provide dedicated windows to represent different parts of the application. For example in a finance application where one part is used to input new transactions while a different user interface is used to change the account details.

Also deploying an application onto different devices, will require an adapted view for each of these devices as they each have specific modalities which might result in a specific interface. For example, as the display screen of a pda is smaller than the screen of a desktop computer, imagine the user interface on the pda to be split into smaller windows through which the user navigates rather than showing all the information in one window as is done on the desktop computer.

Typically different views implies the visualisation of the interface to change. However, providing a different view of an application is more than providing for a different visualisation. Different types of users for instance require different behaviour to be available and therefore different links between the UI and the application to be used. For example, the application behaviour behind the divide button of the calculator is different for the standard and simple calculator.

3.2.3 Applying Dynamic UI Changes

On the verge of new software challenges, dynamic UI changes are of importance. For instance consider an AmI example in which payment transactions to a vending machine are possible from your mobile phone. Assume the payment requires the phone to connect to your banking institution, but this can only be allowed from within the secure area of the vending machine. Hence, if the phone is near the vending machine a payment can be made, but if you are out of communication range, it is not possible. The behaviour for the payment application depends on the location, and therefore context, with respect to the vending machine. As the behaviour of the application changes, the UI changes as well as it offers more or less possibilities. The UI visualisation and behaviour are updated, which means a new view is offered to the user, depending on a change in the application.

Context change requires some properties of the UI to be updated. Dynamic UI changes include changing component properties, layout positioning and triggering other behaviour upon an event. In current software engineering practices this requires the programmer to provide the necessary code to achieve these dynamic UI changes. In .NET and Java this code is part of the event handling code, while in VisualWorks Smalltalk and Cocoa a separate entity is used, namely an applicationModel or controller object respectively. Either way, implementing the necessary changes or providing for the necessary adhoc mechanisms, is what makes providing for dynamic UI changes a cumbersome task for developers.

First of all, current solutions offer little support for applying UI changes. UI component properties can be changed but require knowledge of the actual low-level implementation functions to call. On top of that layout changes are generally either limited to predefined structures (such as grids and boxes), or completely left open to the developers in which case they are responsible for calculating layout positions at runtime.

Secondly, context changes require some control logic to decide what dynamic UI changes to apply at what point. Combinations of context result in combinations of UI changes and cumbersome control logic is introduced. Furthermore, as this control logic combines context knowledge (or business knowledge) with UI knowledge, both concerns become entangled and vulnerable to errors introduced during evolution phases. Current research on context-oriented programming, for example the work with ContextL [Cos05, Hir08], deals with contexts by using a layered approach to activate and deactivate behavioural changes. Although it does not solve the lack of support for specifying UI concerns in separation of one another, it aids in combining different contexts and the behaviour that comes with those contexts.

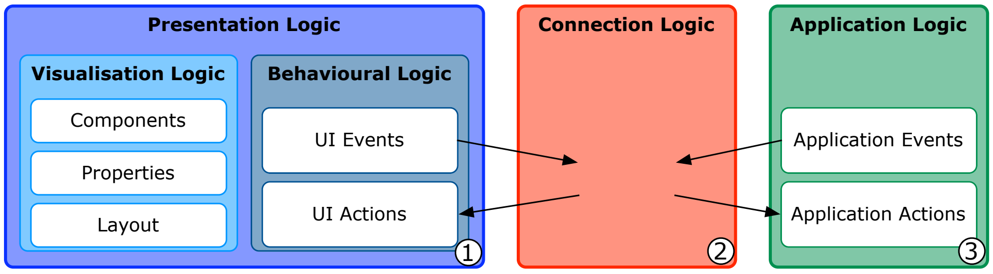


Figure 3.2: User interface concerns

3.3 Key Concepts: User Interface Concerns

Existing approaches that deal with separation of concerns in user interfaces all split up a UI into similar concerns, albeit not always denoted with the same name. For instance in the MVC pattern ‘view’ refers to what the UI looks like[Ree79a], while in model-based UIs this is called ‘presentation’ [Sze96]. Also tiered architectures aim to separate the user interface from the data layer by using an intermediate layer to establish a connection between the two. Conceptually these different approaches split the user interface into similar concerns, but in practice the actual separation happens at different spots, or does not happen at all. When confronted with separation of concerns in user interfaces, people often assume they are talking about the same concern and do not realise they attribute different details to that concern. As we want to avoid confusion during the discussion of user interface concerns, we introduce in this section the terminology we employ in the remainder of this dissertation when referring to user interface concerns. We distinguish between the following concerns, as presented in [God05a, God07b]: Presentation logic, Application Logic and Connection logic. As depicted in Figure 3.2, the presentation logic is split into a visualisation and a behavioural part. Note that the terminology we introduce is similar to the conceptual separation of the other existing approaches. Nevertheless our experiences led to a different connotation for some of the concerns as we particularly focus on the collaboration between UI and application. It is this collaboration that is often the cause of code tangling and scattering. Consequently a one-to-one mapping to the existing terminology should not be assumed as will be discussed at the end of this section.

3.3.1 Presentation Logic

Presentation Logic covers both the visualisation and behaviour aspects of the UI (Figure 3.2 number 1). In essence it covers all elements that are related to the user interface as such.

Visualisation Logic

Visualisation refers to the components that make up the UI and their visual properties. For instance, the calculator in Figure 3.1a, contains buttons for digits, a textfield for displaying the calculated result, and buttons for operators. The visual properties of these components include colour, visibility, layout and state. For example, in Figure 3.1b the `equals` button is positioned at the bottom of the window, spans the entire width of the window and is coloured green. Its font is white and the button is enabled. This means it can be clicked, in contrast to disabled buttons such as the zero button.

Visualisation Logic *specifies what components are part of the UI together with their visual properties and layout.*

Behavioural Logic

The behaviour of a user interface relates to the changes that can take place in the UI and consists of user interface actions and user interface events. *User interface actions* are changes that can happen in the UI, such as visual properties that change, components that are added or removed from the UI, navigational flow that changes. Some of these changes are related to the logic within widgets, like there is enabling a button, making an input-field invisible, changing the colour of a label, setting the background colour of the window, etc. Other UI changes affect the UI as a whole by opening new windows, changing navigation (window) flow, changing master-slave relations between windows, etc. For example in the calculator, clicking the divide button results in changing the colour of certain UI buttons and enables or disables others. Hence the properties of these buttons change. Additionally user interface actions also include opening and closing (slave) windows. For example, imagine the calculator to open a new window with a list component to keep track of all calculations made with the calculator. When closing the calculator, this additional window is also closed automatically.

User interface events are events that are triggered from within the UI, such as clicking the divide button in the calculator, closing the main calculator window, etc. When such an event happens, the UI will invoke specific behaviour. A lot of approaches use event-handlers to deal with UI events. Upon such a UI event, the handler is triggered. This handler specifies what behaviour to execute upon the event. This behaviour consist of UI actions as described above, and application behaviour. For instance, clicking the equals button in the calculator will result in an update of the visual properties of some components (UI actions) and computing the actual result (application behaviour).

Behavioural Logic *specifies the behaviour of a UI by specifying UI actions and UI events. UI actions specify the actual UI changes, while UI events specify on what occasions either or both UI and application behaviour is triggered.*

3.3.2 Application Logic

A second UI concern is the application logic (Figure 3.2 number 3). This concern specifies ‘hooks’ in the underlying code at which the application and the UI will call upon each other. When the application is called by the UI in order to executing certain behaviour or retrieving information, we speak of application actions. When an event in the application triggers a call towards the UI, we use the term application events. Application logic should not be confused with the application. Whereas the application is completely oblivious of its user interface, the application logic provides the link between the UI and the application. As such, it is a user interface concern and therefore part of the UI specification.

Application Actions

Application actions describe what application behaviour (methods) can be called by certain UI events. An example is calling the compute message upon clicking the equals button in the calculator example. Application actions are also used to compute or retrieve the value that should be displayed by a UI component. For instance, this is the case when the display field in the calculator shows the computed result. Hence, the computed result is retrieved from the underlying application. Note however that showing the actual value itself in the UI component is a property change of that component and therefore part of the visualisation logic. In addition, it is possible that before showing the information retrieved from the application, it has to transformed. For example, if the calculator application itself is a decimal calculator, but the user interface shows binary information, the decimal value (retrieved from the application) should be transformed into a binary value (which can be shown by the user interface). Hence, the application action will retrieve this value from the application and transform the decimal result into a binary result. Again, updating the actual UI component with this binary result, is part of the visualisation logic. The application logic is only responsible for retrieving the information and transforming it into a valid value.

Application Events

Application events specify the places in the application where the application calls upon the UI. For instance, when some application data changed and the UI needs to be updated in order to reflect this data change. As an illustration, changing the result in the calculator application will require the display in its UI counterpart to be updated. Application events are of importance as sometimes application data changes from within another view or from outside the user interface altogether. When this change has to be reflected in the interface, the application is responsible for updating the interface. The locations in the application where changes occur with a possible effect on the UI, are application events. Note that the underlying application remains oblivious of its UI but the interactions with the UI are part of the UI logic and is therefore a UI concern.

Application Logic *specifies how the application and the UI call upon each other by specifying application actions and events. Application actions refer to the application behaviour that can be called from within the UI. Application events refer to events in the application that are possibly reflected in the UI.*

3.3.3 Connection Logic

Finally, application and UI events and actions are brought together through the connection logic (Figure 3.2 number 2). As discussed above, both UI behavioural logic and application logic specify events and actions. The events are phenomena that trigger actions. Both UI and application events can trigger either or both UI and application actions. However, application events that trigger application actions only, are not part of the UI concerns and are not considered here.

As an event can trigger actions in both the UI behaviour and application behaviour, the connection between the two is specified in the connection logic. By doing so, connection logic brings presentation logic and application logic together. For instance, clicking the equals button in the calculator (UI event) triggers the compute method (application action) and disables the equals button (UI action). Changing the computed value of the application (application event) updates the display component of the UI (UI action).

As application and UI have to collaborate, otherwise a user interface would make no sense, they have to be connected with one another. This typically leads to code tangling. At some point tangling presentation logic and application logic can not be avoided, but when putting the tangling into a separate concern, being the connection logic concern, its effects are limited and it can be dealt with more easily by developers. First of all because the presentation logic and application logic remain obliviousness of one another. Secondly because the connection logic functions as an abstraction layer in-between the two.

Connection Logic *connects UI and application by linking UI events with UI and/or application actions on the hand, and by linking application events with UI actions on the other hand.*

3.3.4 Analogy with Existing Terminology

Although the terminology we introduced is similar to the terminology used in other approaches for separating concerns in user interfaces, the connotations we have attributed to the concerns slightly differ. Note that even the existing approaches themselves use different connotations or are even perceived differently by their different users, as is often the case with MVC.

In MVC, the model corresponds to the underlying application for which a UI is built. This is what we call the application and is not considered as a user interface concern. The places at which an application can connect to the UI are what we will call application logic. The actual connection is the connection logic. Note that in MVC the places to connect, the actual connection are, together with the UI behaviour, part of the controller. Therefore this controller component contains different UI concerns that remain tangled and closely coupled. The view component of MVC only addresses the visualisation aspect, which is just a part of the presentation concern.

XML based approaches, like XUL and UIML, focus on splitting up the presentation concern. They subdivide this concern into its actual content (i.e. used UI components) and the appearance of these components (i.e. their properties). We consider both these aspects to be part of the visualisation concern. UIML adds a behaviour part to this which provides for the link from UI to the underlying application. This link is what we will call the UI behaviour concern. Hence, while the presentation concern is addressed, the application and connection logic are omitted.

3.4 A Solution for Separating UI Concerns

In order to achieve a better separation of concerns in user interfaces, developers must be offered support in two areas. First of all, support is needed for specifying the concerns in separation from each other. Secondly, all concerns need to be assembled into the final running application. With respect to these two areas, we observe several requirements which a solution should adhere to in order to achieve a better separation. This work was originally presented in [God07a, God07b] but has been refined.

3.4.1 A Separate Specification for Each Concern

In order for evolving and maintaining an application and its user interface, developers need a clear comprehension of the role of the different parts of the system and how these parts work together. Dealing with the several concerns in separation is a first step towards better comprehension [Tar99]. Specifying a UI concern in separation from the others, enables changes in isolation and hence supports the evolution of a concern independently from others. For example when resizing the buttons in the standard calculator from Figure 3.1, this changes the presentation concern but does not affect the application logic nor the connection logic.

Furthermore a separate specification for each concern implies that the specification of one concern can be reused for other UIs. For instance, imagine the calculator application to run on both a desktop computer and a PDA. In this case the presentation concern will change because the UI components ‘look’ different on different devices. As the application remains the same, so will the application specific concern that describes where the application and UI call upon each other. In order to support developers in creating UIs, a solution should fulfill the following requirement:

Requirement 1: A solution for supporting separation of user interface concerns needs to provide support for creating *a separate specification for each concern.*

3.4.2 High-Level Specifications

When linking the application with the UI, developers are currently confronted with the low-level implementation of this connection. Also for updating component properties and positioning they often need to know how to achieve this and especially for dynamic UI changes developers need to provide the necessary program statements that actually update the UI. For example in Figure 3.1 the number buttons in the standard calculator are ranked differently than the numbers in the simple calculator. When changing between calculator modes, the number buttons have to be repositioned. The developers need to provide the code that calculates the new layout positions and updates the number buttons.

As developers are still confronted with these low-level technicalities of the several UI concerns, they lose an overall view of the system and its behaviour. Using abstractions empowers the developer to focus on 'domain concepts' and what the system has to do, rather than having to deal with implementation details. Therefore, it is advisable for a solution that developers do not have to deal with the low-level implementation of the concerns but can focus on a high-level specification for the several concerns. This will add to a better comprehension of the overall system. For example, instead of having to recalculate the positions of the number buttons it is more straightforward to specify that the number buttons 1,2 and 3 are displayed on the same row instead of being shown in one column.

Requirement 2: A solution for supporting separation of user interface concerns needs to provide support for creating *specifications at a high-level of abstraction.*

3.4.3 Mapping High-Level Entities onto Code Level Entities

Since the concerns are described in separate specifications, we need a mechanism that composes them into a functional application. The final application is obtained by mapping the high-level UI specification onto low-level code entities. Such a low-level application is specific for the device or platform it is running on. For instance, a widget in a Smalltalk UI is different from a component in an XML UI. As a consequence, a mapping will be specific for certain low-level components, but holds for all high-level UIs mapping onto that same low-level device or platform. On the other hand a high-level specification can be reused to translate to different low-level applications by using the different corresponding mappings. Once such a mapping has been established, typically high-level UI developers will reuse these 'libraries' and not have to deal with the mapping specifications as such. For instance in UIML, the UI descriptions are transformed into an actual UI by a renderer which knows how to create the low-level UI widgets.

Selecting a mapping not only depends on the kind of device or platform that is targeted, but can also depend on other context data. For instance, the alternative calculator in Figure 3.1b uses colourful buttons instead of the standard grey buttons. Consequently, the high-level digit buttons are transformed into different low-level actual buttons. This appearance information is part of a UI description but it can be filtered out. This style information is also in UIML and XUL separated from the actual component information and contents, although all alternatives for a certain context are still put within one description file. In addition, the UI renderer will decide upon generating the UI what context to apply, which disallows contexts to depend upon dynamic information.

In summary, a mapping can be reused among several high-level UIs in order to map these UIs onto the same low-level code entities (i.e. device specific). Alternatively one high-level UI specification can be reused for different devices by applying the corresponding mapping for that device. As such, the high-level UI is transformed into different low-level device specific UIs. Hence, using mappings and a mapping mechanism avoids a tight coupling between the high-level specifications and the actual UI primitives.

Requirement 3: A solution for supporting separation of user interface concerns needs to provide *a reusable mapping from high-level UI entities onto code-level entities.*

3.4.4 Automatically Composing the Different UI Concerns

As stated by requirement 2, it is advisable to hide the low-level technicalities from the developers. This is especially true for the technicalities of the composition mechanism that combines the concerns into the final application. Current solutions often require the developers to provide this mechanism themselves and by doing so these solutions annul the benefits of having separated the concerns in the first place. Developers need to re-introduce complexity to combine the UI specification with the underlying application. For instance, when using the observer mechanism in Visualworks Smalltalk, they need to add notify messages to the model. The event handling system in Java requires linking components with event handlers and providing event handling code, which in its turn might need to address both model and view. In addition, automatic composition mechanisms that still require fine-tuning the resulting code afterwards are also insufficient for supporting developers in creating and evolving user interfaces. As the resulting code is obviously tangled, developers should, in evolution phases, be able to work on the original (separated) concerns at all times, without needing to go through the fine-tuning process again. In short, the composition mechanism to assemble the UI concerns into a final application should be provided and be applied automatically, without exposing developers to the resulting code afterwards.

Requirement 4: A solution for supporting separation of user interface concerns needs to provide *an automated way to compose the different UI concerns and the underlying application into a complete final software system.*

3.4.5 Support for Automated Layout

Evolving a system includes adding and removing components from the UI. As this has an impact on the visualisation of the UI, one needs to 'redraw' the UI accordingly or specify the new position for components. The best alternative to manual positioning is to group components and have their positions 'shift' automatically. For instance, layout managers in Java allow putting components into a box or a grid, which will be filled up with components. The positions of these components shift automatically when a component is removed. However, this mechanism fails when evolution implies that positions of components change more drastically. For example, repositioning labels from 'the left of an input field' to 'above the input field'. The problem of positioning components by hand worsens when UIs change because of context changes. For each context, and for each combination of contexts, developers would have to provide the corresponding component positions. Especially when changing the UI programmatically, developers also have to implement the layout update mechanisms. Both changing positions by hand, as well as calculating new positions programmatically, pose a huge burden on the programmers and take them back to a low-level view, away from the high-level abstractions that introduced the necessary comprehension for evolving and maintaining a UI more easily. Therefore, ideally a UI visualisation is expressed at a high-level and is automatically translated into a low-level layout of individual components.

Requirement 5: A solution for supporting separation of user interface concerns needs to provide support for *automated layout.*

3.4.6 Conclusion

We have postulated five requirements which a solution should adhere to in order to achieve a separation of concerns for user interface code in software systems where application and UI link both ways, multiple views can exist and dynamic UI changes are required.

Table 3.1: Comparison of Existing approaches for Separation of Concerns in User Interfaces

		MVC	Model-based Uis	UIML & UsiXML	Adam&Eve
Req. 1	separate	+	+	+	+
	each concern	-	∼	-	-
Req. 2	high-level	-	+	∼	-
Req. 3	mapping	-	∼	-	-
Req. 4	composition	-	∼	∼	∼
Req. 5	layout	-	-	∼	∼

As a conclusion we present an overview in table 3.1 of current solutions for separating user interface concerns and how they live up to these five requirements. Note that in traditional business applications typically MVC is used for separating user interface code from application code. Both Model-based UIs and user interface description languages like UIML and UsiXML, focus on generating user interface code based on models or 'declarative' descriptions. Providing the transformations from models or descriptions to generated code, is left for the developers. Current declarative solutions such as Adobe's Adam&Eve tackle only part of the user interface concerns. Typically these solutions deal with expressing layout.

3.5 Conceptual Methodology

Figure 3.3 depicts the process which developers go through when using such a solution. Note that we discuss the implementation process of a UI and do not consider how the UI should be modelled. Nevertheless quite some models (e.g. the ones used in model-based UIs) can be mapped onto the separate concerns specifications proposed in this dissertation. The core application is developed in separation from the user interface

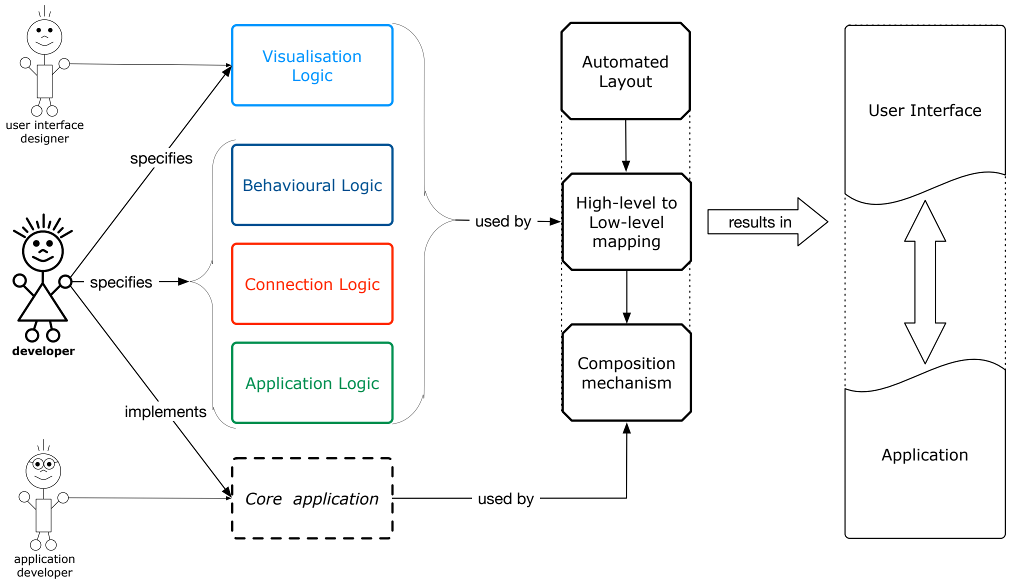


Figure 3.3: A solution for separating user interfaces concerns

specification. This UI specification consists of its visualisation and its functionality, namely the UI behaviour, how it hooks into the application (application logic) and how events in either application or UI trigger actions in either application or UI (connection logic). The provided solution aids developers by taking both the UI specification and the application as input and producing the running application as output. To do so it has to provide a mapping from high-level entities to low-level entities, support for automated layout and a composition mechanism that composes the actual system.

When providing a better separation of user interface concerns, it is more straightforward that different experts are responsible for different concerns. For instance the core application is developed in complete obliviousness from the user interface. In addition, a graphical designer can focus on the visualisation of the UI which now is expressed at a level of abstraction that obscures the designer from practical low level technicalities.

The conceptual solution we discussed in this chapter, will be put to practice in the following chapters. In Chapter 4 we introduce DEUCE as an architecture for Declarative User Interface Concerns extrication and we show how it is used from the developers' point of view. Chapter 5 gives the details of the mechanisms used behind the scenes such that DEUCE supports high-level specifications, mapping from high to low-level entities, automated layout and automatic composition of the concerns and the application into a running system.

3.6 Conclusion

Although current solutions exist for separating concerns in user interfaces, they fail in fully supporting the developers when implementing an application and its user interface. The problem becomes even more apparent as new software challenges increase the need of flexibility of user interfaces. Solutions in achieving a separation of user interface concerns should aid the programmer in developing software systems where

- application and UI are linked in both ways,
- different views on the same application are supported, and
- dynamic UI changes are applicable.

Furthermore the separation of concerns in user interfaces should be carried through all the way to the implementation level. By doing so the separation can be maintained during future evolution phases.

Such a separation of concerns is achieved by the conceptual solution we proposed in this chapter. First of all we defined the terminology we use throughout this dissertation with respect to the various user interface concerns. These definitions are of importance as they put the reader on the right line for the rest of this dissertation. Note that it is sometimes forgotten that the application logic and connection logic concern are as significant as the presentation logic concern.

In order to support a programmer in creating flexible user interfaces, and to benefit from a full separation of concerns, a solution for separating user interface concerns should adhere to the following requirements:

- **Requirement 1:** *A separate specification for each concern.*
- **Requirement 2:** *Specifications at a high-level of abstraction.*
- **Requirement 3:** *A mapping from high-level entities onto code-level entities.*
- **Requirement 4:** *An automated way to compose the different UI concerns and the underlying application into a complete final software system.*

- **Requirement 5:** *Provision for automated layout.*

We elaborated on each of these requirements. In the next chapter we will put them into practice by providing a proof-of-concept implementation for the conceptual solution proposed in this chapter.

Chapter 4
DEUCE: A Proof-of-concept for Separating User Interface Concerns

In Chapter 3 we discussed a conceptual solution for separating concerns in user interfaces by providing a set of requirements that should be fulfilled in order to achieve this separation of concerns. In short, a solution should provide for a separate specification for each concern, allow for high-level specifications and a mapping onto low-level code entities, provide an automated way to assemble the different UI concerns into a complete final application, and offer support for automated layout.

In this chapter we will introduce DEUCE as an instantiation of the conceptual solution. We start by motivating the choice for a declarative approach for expressing UI concerns in Section 4.1. Next we elaborate on the declarative meta programming language that was chosen to implement this declarative approach. Declarative meta programming uses a declarative programming language at the meta level to reason about or to manipulate programs in some base language. More specifically, DEUCE uses a logic language (SOUL) on top of an object-oriented language (Smalltalk). Developers specify the user interface at a high-level with declarative statements in SOUL. The underlying application, both data and business logic, are developed in Smalltalk. DEUCE assembles the several UI concerns and the application into a final application with an interface, which is be created in Smalltalk, but maintains a link to the declarative UI specification. After introducing SOUL in Section 4.2 we show how developers use DEUCE to express the UI concerns in Section 4.3. We end this Chapter with a small case study in Section 4.4. In this case study we will extend the example from Section 4.3 to illustrate how a user interface specification in DEUCE is evolved.

4.1 A Declarative Approach

When it comes to specifying user interfaces, quite some UI creation tools have adopted a *declarative* approach for doing so (see Chapter 2). The major benefit is that one focusses on what to achieve, rather than how to achieve it. Also for user interfaces it is intuitive to express what a UI looks like, rather than expressing how that look is established.

For instance, it is easier to express that a component is positioned to the left of another component than to express how to achieve this positioning.

For instance, graphical editor tools are often used for creating user interfaces and can be seen as way to declare a UI. The UI developers or designers put components visually on a screen and fills out forms to specify component properties. The tools are responsible for transforming these graphical representations into a lower-level code representation such that it can be used within an application. Other approaches such as UIML and XUL, provide XML-like specifications of what components are part of the UI and what their properties are. Additionally UIML uses a declarative layout description for positioning and aligning UI components. Both JGoodies for Java Swing and Adobe's Eve (layout library) [Par07] use declarative descriptions to describe layout. Adobe's Adam (property model library) declares relationships on a collection of values to link UI components with values.

All these approaches use declarative descriptions at some point to describe parts of the UI concerns. It is intuitive for developers to think about the UI in terms of what it represents and declarative specifications will increase his understanding of the system. Hence we formulate the following consideration to take into account when providing a solution towards separation of UI concerns:

Consideration 1: *The solution for achieving separation of concerns in user interfaces should consider using a declarative formalism to express the UI concerns.*

Furthermore, using the same underlying formalism to express all the UI concerns facilitates putting the several concerns together into a final application. Obviously dedicated tools can be built on top of this to offer the high-level developers the special purpose formalism of his choice for expressing each of the concerns. The low-level developers who are responsible for putting all the concerns together, will benefit from using this uniform medium though. Therefore,

Consideration 2: *The solution for achieving separation of concerns in user interfaces should consider to use one uniform medium to express all UI concerns.*

The other requirements as defined in Chapter 3 can benefit from these considerations. First of all, in a declarative approach, a mapping between the high-level entities declaration and the low-level entities' declaration will consist of a set of rules. Switching to another mapping means that a different rule set becomes applicable. In addition, extra knowledge can be added to a mapping itself for deciding when to translate to a certain low-level entity or an alternative thereof. The declarative reasoning mechanism provides the decision mechanism to choose the right mapping. Secondly, providing an automated composition mechanism means that the mappings (see Section 3.4.3) are applied automatically. This includes selecting the right mapping rule from several alternatives, for instance when these mappings depend on context. In a declarative medium, these alternative mappings correspond to alternative rules. In declarative programming, several alternatives for one rule can exist and if one fails, another one is tried. Note that the composition mechanism is also responsible for putting the right glue code (to link UI and application) in place. Finally, other approaches have already benefit from declarative layout descriptions. For instance both JGoodies for Java Swing and UIML use a declarative layout description for positioning and aligning UI components. Components are placed in a grid structure which is built from vertical and horizontal boxes. Adobe's Eve library does the same by using rows and columns. These declarative descriptions put layout relations at a higher level. Basic relations are putting one component above or next-to another one. For instance in Figure 3.1a, the digit 1 is positioned next to digit 2. More advanced relations position a group of components in one column or row, or automatically fill up a certain amount of columns or rows. For instance, in Figure 3.1a the digits 1, 2, 3 are positioned in one row. Or the digits 1 to 9 are spread over three columns. Even when components are removed, the layout relations will hold and adapt the positioning of the other components without leaving gaps where a component was removed. For instance, when removing the digit 2 in Figure 3.1a, the one row rule will put digit 3 next to digit 1 without leaving a gap for the digit 2 button.

4.2 Smalltalk Open Unification Language

The Smalltalk Open Unification Language (SOUL) [Wuy01] is a logic programming language, similar to Prolog, built on top of Visualworks Smalltalk. SOUL works in symbiosis with Smalltalk which allows for interacting directly with Smalltalk objects. This facilitates to reason about and manipulate Smalltalk programs. For instance, it allows accessing and changing Smalltalk user interface objects directly from within the logic level. This is achieved by allowing Smalltalk statements to be used at the logic level and to be parametrised with logic variables such that Smalltalk code can be executed during the verification of logic queries. In what follows we introduce both SOUL's basic logic features and its particular SOUL features.

4.2.1 Logic Programming

As SOUL is a logic programming language, we first discuss its 'standard' logic programming basis. Logic programming is a declarative paradigm that was developed in the seventies in the domain of knowledge systems. A program is declarative if it describes what to achieve rather than how to achieve it. Thus a logic program concentrates on a declarative specification of what the problem is, and not on a procedural specification of how the problem is to be solved. Rather than viewing a computer program as a step-by-step description of an algorithm (as in traditional languages), the program is thought of as a logical theory and a procedure call is viewed as a theorem the truth of which needs to be established. The execution of a program comes down to searching for a proof. In order to do so, the database of a logic program consists of facts and rules which are consulted by queries.

- *Facts* hold static information that is always true in the application domain.
- *Rules* derive new facts from existing ones. The conditional part of the rule should be true in order to conclude the premise of the rule. Rules, also called predicates, consist of several goals. The premise is made up out of one goal, whereas the conclusion can consist of one or more goals.
- *Queries* are used to access the data in the database. Finding an answer to such a query is carried out by matching it with facts (initial or derived).

In essence, finding a match is proving that a statement follows logically from some other statements. This reasoning process is also called resolution and adds a procedural interpretation to logical formulas, besides their declarative interpretation. Because of this procedural interpretation, logic programming can be used as a programming language. Kowalski's equation "algorithm = logic + control" [Kow79] also notes this. In this equation, logic refers to the declarative meaning of logical formulas and control refers to the procedural interpretation.

The resolution mechanism will try to verify a query with respect to the set of rules and facts. If a query can be verified and is found valid, the output of the process will be success. At that point, unbound variables in the initial query are possibly unified (two-way pattern matched) with a valid value. Otherwise, the output of the process is failure. Upon failure, alternative rules for the predicate are triggered. These alternatives are provided by the programmer. The reasoning process will continue triggering rules until one succeeds or all fail. Note that, as in regular Prolog, a cut (denoted with `!`) in a succeeding rule stops the reasoning process looking for other solutions, and hence triggering alternative rules.

4.2.2 Prolog Expressions in SOUL

Logic expressions in SOUL use a slightly different syntax from their Prolog counterpart. Variables start with a `?` instead of a capital letter. Lists are denoted with `< >` instead of `[ ]`. The head and conclusion of a rule are separated with an `if` keyword instead of `:-`. The wildcard variable _ which does not bind to any values, is represented by a single `?`. For example, a fact in SOUL can be expressed as follows:

```
usedComponentsInInterface(test, <one,two,three>)
```

The predicate is called `usedComponentsInInterface`. It has two arguments, the first one of which is a symbol `test` and the second one a list (denoted by `< >`) with elements `one`, `two`, `three`. The example rule shown next uses this predicate as a one of its goals in the conclusion on line 2. When verifying this goal, the variable `?interface` is unified with `test` and the variable `?comps` with the list `<one, two, three>`.

```
1 isComponentInInterface(?component,?interface) if
2    usedComponentsInInterface(?interface,?comps),
3    includes(?component,?comps)
```

The variable ?comps, now bound to the value <one, two, three>, is used for verifying the next goal. The resolution mechanism will try to unify the variable ?component. The predicate includes binds this variable to one of the values in the list. If the variable ?component was unbound upon calling the isComponentInInterface predicate, the rule will succeed when binding any value of the list. If the ?component value was bound upon calling the predicate, its value will have to unify with one of the values in the list in order for the rule to succeed. The includes predicate on line 3 is one of the predicates often provided by a logic language. SOUL also has a database with such common logic predicates [De 02]. We will not give further details about these predicates as they are not part of DEUCE itself but of the logic language being used.

4.2.3 Smalltalk Blocks at the SOUL Level

SOUL uses Smalltalk blocks (denoted by [ ]) to incorporate Smalltalk expressions and values at the logic level. Such a block is evaluated at verification time and can be parametrised with logic variables. Since the reasoning process binds these variables to values at runtime, they can be used to determine what object a Smalltalk message is sent to. For instance in the following code snippet, Smalltalk blocks are used to retrieve a component from a running user interface and to enable that component.

```
1  enable(?componentName,?userInterface) if
2     equals(?component,[?userInterface componentAt: ?componentName]),
3     [?component enable. true]
```

Upon evaluating the Smalltalk block, all the variables used in the block should be bound to a value. Hence, the variables userInterface and ?componentName used on line 2 should be bound when verifying this goal. The Smalltalk block in this goal is used to retrieve the component with as name the value bound to ?componentName from the Smalltalk user interface bound to the variable ?userInterface. The result of this Smalltalk message is a UI component (i.e. a Smalltalk object). This result is unified (and therefore bound) with the ?component variable through the equals predicate. If a block is used as a goal on its own, as illustrated on line 3, it needs to return a boolean value since a logic goal always evaluates to true or false (line 3). In this goal the object bound to the ?component variable is sent the enable message. As this Smalltalk message does not necessarily return a true value, the block needs to return it explicitly.

Note that Smalltalk blocks at the logic level allow SOUL to access Smalltalk objects directly. It can be used in combination with Smalltalk's meta-level protocol to reason about and change Smalltalk programs from within SOUL. SOUL provides a library, called LiCoR, with predefined predicates to do so. DEUCE provides its own library to manipulate Smalltalk user interfaces and to link the interface with the application.

4.2.4 Variable Quantification

When proving a predicate, the reasoning process will trigger all alternatives that are provided for that predicate. However, sometimes this process can be optimised when it is known whether a variable is already bound or not. For instance, in the following example when the `?component` variable is unbound, the goal on line 2 will bind it to a valid Smalltalk component object before the goal on line 3 retrieves its name from its spec by sending it a Smalltalk message. If the `?component` variable is bound however upon calling this rule, the goal on line 2 is obsolete and the reasoning process can be sped up by omitting it.

```
1  name(?component,?componentName) if
2     isComponentIn(?component,?ui),
3     equals(?componentName,[?component spec name])
```

Prolog provides two meta-logical predicates to verify whether a variable is already instantiated or not, namely `var` and `nonvar`. `var(?x)` will succeed if `?x` is an uninstantiated variable and does not have a value assigned yet. `nonvar(?x)` succeeds if `?x` is instantiated, and thus already has a value assigned. The short version in SOUL allows to quantify variables in this way with a `+` or `-` sign respectively. For instance, in the following code example, if both `?component` and `?layout` variables are already bound to a value, the first rule will never be triggered but the second rule will.

```
1  layout(+?component,-?layout) if
2     equals(?layout,[?component layout])

3  layout(+?component,+?layout) if
4     [?component layout: ?layout. true]
```

In SOUL (and DEUCE) this mechanism is used to determine what message to send to a Smalltalk object. Logical rules work two-ways, such that they are valid no matter what variables are bound or unbound. However, as logical variables used in Smalltalk blocks have to be bound before a Smalltalk message can be sent, rules with Smalltalk blocks usually do not work two-ways. For instance, if the `?component` variable on line 2 is not bound, the Smalltalk message `layout` would be sent to `nil` and result in an error. By using the variable quantification mechanism, several alternatives for a predicate can make it work two ways. The first `layout` rule on line 1 is called with an unbound `?layout` variable. By sending the Smalltalk getter message `layout` to the component bound to `?component`, its layout is retrieved and bound to the `?layout` variable through the equals predicate (line 2). If the `?layout` variable is bound, as in the rule on line 3, it implies sending the Smalltalk setter message `layout:` to set the component's layout to this new value. As there are no logic variables to be bound in this second predicate, its goal is merely evaluating the Smalltalk block. Recall that a goal should either be true or false, thus the Smalltalk block needs to return a boolean value. In conclusion, in the example above the variable quantification mechanism of SOUL is used to determine whether the Smalltalk getter or setter method is to be sent to the component. It turns the `layout(+?component, ?layout)` into a two-way predicate with respect to the `layout` variable.

4.2.5 Repositories and Repository Variables

In SOUL, logic facts and rules are stored in static logic databases, called layers. In order to use them at runtime, these layers are added to a runtime logic database, called a repository.

Upon instantiating a repository, its layers are loaded into its knowledge base by adding all the predicates of that layer to the knowledge base of the repository. Once instantiated, the reasoning process uses this knowledge base to look up information (predicates and facts) when solving queries. In addition, extra information can be asserted (added) or retracted (removed) during this process. However, when the repository is re-instantiated, or in other words rebuilt from scratch from its layers, the runtime knowledge base is reset and all previous information that was asserted or retracted from it, is lost. Repositories are re-instantiated when one of their layers is updated. As one layer can be part of different repositories, changing it will re-instantiate all the repositories it belongs to.

When starting the reasoning process, one can specify what repository to use as a knowledge base. Rules and facts defined in other repositories are out of scope at that point and can not be triggered during reasoning. In order to access these repositories nevertheless, SOUL allows to specify what repository to use for a certain query. This is done by adding `?repository->` before the query. An example is shown below.

```
1  adjoin(?ui) if
2     above(?compName1,?compName2),
3     componentWithNameIn(?comp1,?compName1,?ui),
4     componentWithNameIn(?comp2,?compName2,?ui),
5     constraintSolver(?ui,?solver),
6     ?layout->adjoin(above,?comp1,?comp2,?solver)
```

The first four goals (lines 2–5) are resolved in the same repository as the one in which the predicate `adjoin(?ui)` is defined. The last goal `adjoin(above, ?comp1, ?comp2, ?solver)` (line 6) is annotated with a repository variable `?layout`. Hence, it is resolved in the repository that is bound to this variable. This variable is the same for the entire repository in which the predicate `adjoin(?ui)` is defined. When replacing this repository by a new one, it is sufficient to rebind the `?layout` repository variable to the new repository without needing to change the `adjoin(?ui)` rule. Note that since SOUL is written in Visualworks Smalltalk, a repository is a Smalltalk object. Its instance variables, and therefore also it repository variables, can be accessed through Smalltalk. Therefore repository variables can be rebound programmatically to other repositories at any time.

4.2.6 Using SOUL for DEUCE

DEUCE will use SOUL as a uniform medium to express all UI concerns declaratively. The three concerns Presentation, Application and Connection logic will be expressed with declarative SOUL statements. SOULs layering and repository mechanism allows to do so at different levels of abstraction. Hence, the first requirement to express the several concerns separately and the second requirement to do so at a high-level op abstraction are fulfilled. This will be illustrated in the next section.

4.3 A Developer's View on DEUCE

Recall from Chapter 3 that in order to achieve a separation of concerns the provided solution should allow to specify the several UI concerns at a high-level and in separation from one another. In this chapter we will show how the developer can use DEUCE to do so. The example used throughout this chapter is the calculator application that was introduced in Section 3.1. Note that although this section shows the declarative statements that specify UI concerns, a set of tools will be provided in the future to describe the UI concerns such that the programmer needs not to be knowledgeable of the specifics of the declarative language. The descriptions created with the tools can be translated into declarative language statements and will not change how DEUCE reasons upon these statements in order to construct the actual interface.

Figure 4.1 depicts the process developers go through when using DEUCE to create user interfaces. The application is developed (number 1) in separation of the user interface. For the user interface, the developer specifies the presentation logic (number 2 and 3), the application logic (number 4) and the connection logic (number 5). How these specifications are used by the DEUCE reasoning engine and its other components to create the running UI (number 6), is explained in Chapter 5.

4.3.1 Presentation logic

The presentation logic concern specifies everything that is related to the actual UI, being both its visualisation aspects (i.e. what the UI looks like) and its behavioural logic (i.e. how the UI behaves). It is specified by the developer in different steps: selecting components (Figure 4.1 number 2), specifying visualisation logic (Figure 4.1 number 3) and specifying behavioural logic (Figure 4.1 number 3).

Visualisation Logic

In order to create a user interface, the programmer provides a set of components available to DEUCE. To do so, the standard Visualworks Smalltalk UI Painter is used to select components by putting them on the canvas (Figure 4.1 number 2), as is shown in Figure 4.2a, and give them unique names. There is no need for the developer to carefully position these components, as the layout will be specified declaratively later on. The set of components is next translated into DEUCE facts that describe these components. Alternatively the developer can describe these facts in DEUCE immediately without the use of the UI Painter. For example the button 1 in the calculator translates into the following facts:

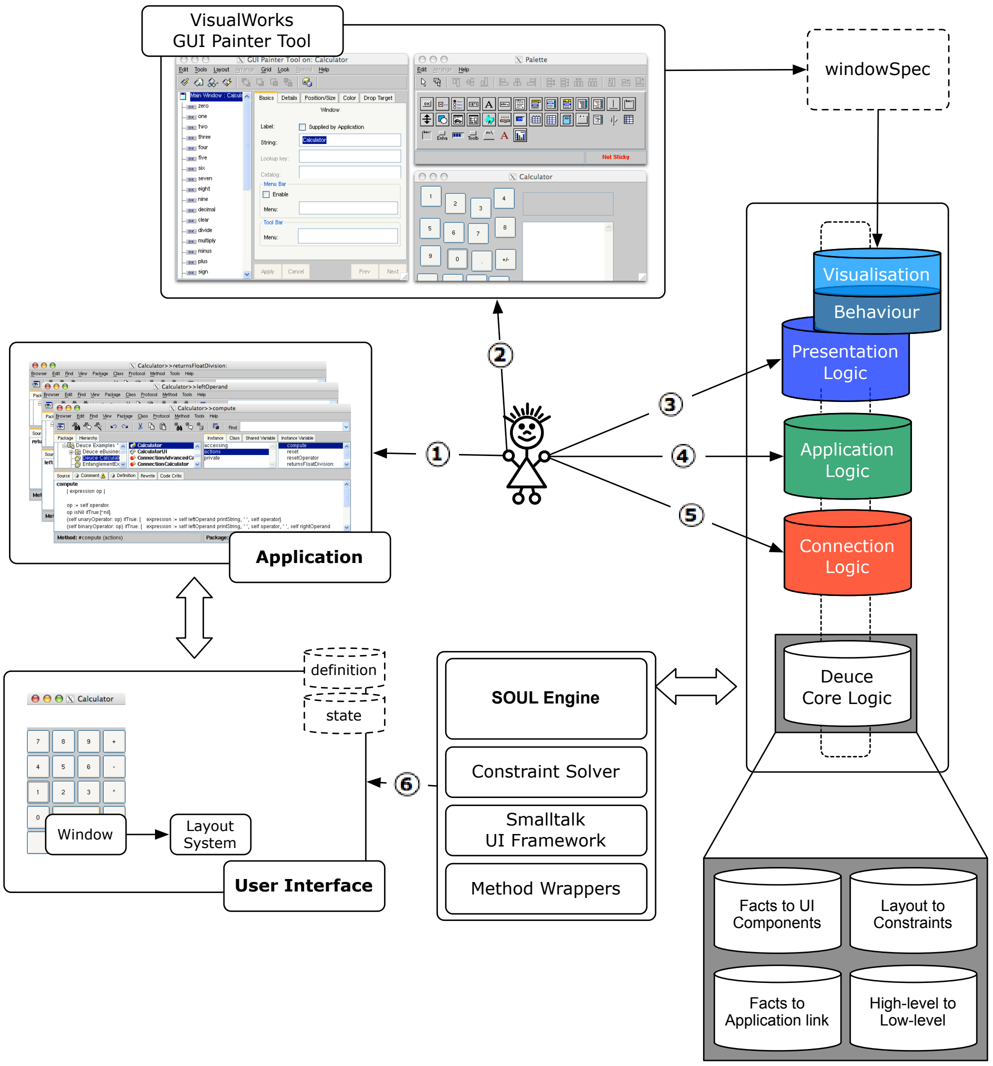


Figure 4.1: Using DEUCE to create user interfaces

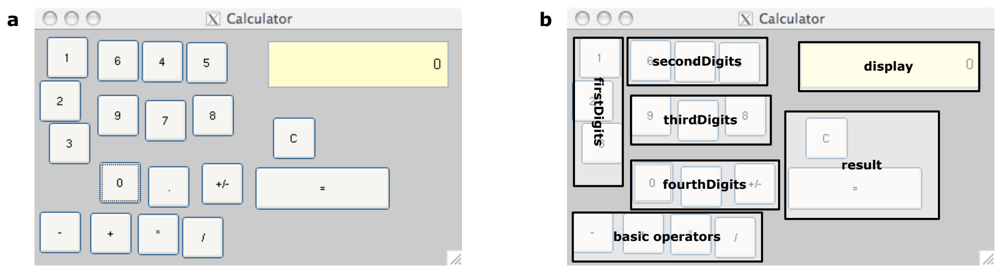


Figure 4.2: Components in the calculator example

```
button(one).
text(one,['1'])
```

When creating a user interface, the programmer first specifies the visualisation part of the presentation concern. This expresses what the interface looks like by specifying what components are part of an interface and what layout relations apply. Also other properties such as size, colour, font, etc. are part of this concern. For instance the user interface of the standard calculator (see Figure 3.1a), has one window with a set of components which is described as:

```
window(standardCalculator).

containsComponent(standardCalculator,zero).
containsComponent(standardCalculator,one).
containsComponent(standardCalculator,two).
containsComponent(standardCalculator,three).
...
containsComponent(standardCalculator,divide).
containsComponent(standardCalculator,clear).
containsComponent(standardCalculator,operatorDisplay).
containsComponent(standardCalculator,numberDisplay)
```

Several windows can be defined for one UI. For example, when extending this UI with extra operator buttons, this can be defined as

```
containsComponent(extraButtons,power).
containsComponent(extraButtons,reciprocal).
containsComponent(extraButtons,percentage)
```

When choosing to add these extra components to the original standardCalculator window of the UI, its components are added and layout is recalculated and applied. Previously defined specifications remain valid while components can be added and removed at runtime.

Components can be grouped together because they belong together visually or logically. Groups can be used in layout relations such that the entire group is positioned with respect to other components. Additionally groups can be used to change properties of all components within the group at once. Some of the groups illustrated in Figure 4.2b, are for instance expressed as:

```
group(firstDigits,<one,two,three>).
group(operators,<plus,minus,multiply,divide>)
```

Groups specifications can also be used to specify what components are used in a window. This would allow for example to replace the previous `containsComponent` rules with:

```
containsComponent(standardCalculator,firstDigits).
containsComponent(standardCalculator,secondDigits).
...
containsComponent(standardCalculator,operators)
```

By specifying component properties the developer sets the colour, font and size of the UI components. For instance, the following specifies that all components in the UI should be 40 by 40 pixels wide, have a grey background colour and a black font text. Recall that `?` denotes a wildcard variable which is not bound to any particular value, and therefore the `minimumHeight` predicate for example applies for all components in the user interface.

```
minimumHeight(?,40).
minimumWidth(?,40).
backgroundColour(?,veryLightGray).
foregroundColour(?,black)
```

These settings can be overruled by adding more specific rules for some of the components, for instance colouring the equals button green and making its font white with the following:

```
backgroundColour(equals,green).
foregroundColour(equals,white)
```

When applying UI specifications these more specific specifications will apply for the equals button, while the general rule from the previous example applies for all other buttons.

Next, the developer describes the layout relations that apply for the components in the calculator. For instance, the components in the group `firstDigits` are positioned on one row by specifying:

```
oneRow(firstDigits)
```

The components in the group `operators` are positioned in one column. One can also put all components in a list in one row or one column, as is done with the fact on line 2 for all groups containing digit buttons:

```
1 oneColumn(operators).
2 oneColumn(<firstDigits,secondDigits,thirdDigits,fourthDigits>)
```

As will be explained in Section 5.3 both the `oneRow` and `oneColumn` relations are based on the basic relations above and `leftOf`. These basic relations can also be used to position the components of the calculator. For instance when specifying that the `fourthDigits` group is positioned next to the `operators` group and the `display` above the `firstDigits` group.

```
leftOf(fourthDigits,operators).
above(display,firstDigits)
```

In addition, in order to adapt the display size such that it spans the whole interface in alignment with the outer components, the following alignment statements are specified:

```
rightAlign(display,plus).
leftAlign(display,one)
```

As the `display` is to be left aligned with the `one` button and right aligned with the `plus` button, its width will span from the left side of the `one` button to the right side of the `plus` button. This concludes the visualisation of the standard calculator.

Within an organisation or company, user interfaces typically follow a set of guidelines to obtain a coherent look throughout the organisation's applications. For instance, guidelines subscribe that labels should be positioned to the left of inputfields, titles are of a certain font and colour, sizes of components are justified and the first component on a window is the organisation's logo. These visualisation specifications (properties and layout), can be combined into a (layout) strategy. Applying such a strategy to all UIs of that organisation, achieves the desired coherent look.

Behavioural Logic

Another part of the presentation concern is the behavioural logic which describes what UI events and actions happen within the user interface. As explained in Section 3.3.1, UI actions are changes that can happen in the UI, such as adding new components and changing component's properties. UI events are events that can be sent to the UI and will trigger behaviour, including UI actions. For example in the standard calculator, division by zero is not allowed. Upon clicking the divide button (i.e. UI event), the zero button is disabled (i.e. action). The underlying application is responsible for not allowing division by zero. However, disabling the zero button is detached from the application since it is merely a change of the enabled property of the button. As a consequence, it is entirely detached from the application and part the presentation logic.

In DEUCE, as part of the presentation logic, the developer specifies that `clicking` the `divide` button triggers the `divideClicked` UI event:

```
UIEvent(divideClicked,divide,click)
```

Specifying this fact de-couples the actual click event on the divide button from its logic counterpart. This allows linking the same logic `divideClicked` event with other actual events on other buttons. For instance, the `divideClicked` event is also triggered when someone types `/` on the keyboard. Hence, one could specify

```
UIEevent(divideClicked,keyboard,/)
```

Upon an event, DEUCE launches a query that specifies the behaviour to be linked with this event. Hence, the developer specifies that the `divideClicked` event triggers the `operatorClicked` query.

```
linkUIEventToQuery(divideClicked,operatorClicked(divide,?ui))
```

Note that the predicate `operatorClicked` takes two arguments, the first of which is bound to the value `divide` and the second is a variable `?ui`. As will be explained in Section 5.8 the latter will be bound upon calling this query with the actual running UI instance at that moment.

The query that is launched, is usually part of the connection logic, as will be illustrated further on. In its turn this query can call UI actions. For example, the `operatorClicked` query, which is part of the connection logic, triggers the `states` rule, which is part of the presentation logic. The following piece of code shows the `states` query for the divide button. In the standard calculator the first rule applies (lines 1–3). In the simplified version of the calculator, this rule is extended by the second rule (lines 5–9). When clicking the divide button in the simplified calculator, the `states` query will be triggered and both these rules will apply, with the most specific rule being called last. Expressing the same behaviour in Smalltalk would require a test to know what kind of calculator is being used and what UI behaviour is inflicted by it. In DEUCE the behaviour requires an additional rule in the specification of the simple calculator. If more specific states apply, more rules will be added but there is no need to test what kind of calculator is being used and hence no need to adapt cumbersome code statements.

```
1  states(divide,?ui) if
2     disable(zero,?ui),
3     disable(decimal,?ui)

4  Extra rule for the Simple Calculator
5  states(divide,?ui) if
6     digitsToEnableUponDivision(?ui,?allowedDigits),
7     allDigits(?ui,?digits),
8      disableComponents(?ui,?digits),
9      enableComponents(?ui,?allowedDigits)
```

In the standard calculator clicking the divide button will disable the `zero` and `decimal` button. Clicking the `divide` button in the simplified version of the calculator has a larger impact, as only buttons that lead to a whole number result after division are to be enabled. therefore, the application will be asked for all allowed digits by solving the goal on line 6. All digit buttons will be disabled (line 8) and all allowed digits will be enabled (line 9). Note that the `states` query is part of the behaviour (presentation) logic because these UI actions affect the UI logic, namely components and their properties.

4.3.2 Application Logic

A user interface is connected with an underlying application that provides the behaviour of the actual application. Events in the UI will trigger this behaviour in the application (i.e. application actions). On the other hand, the application will trigger updates in the UI through application events. The points in the application at which the application and UI can be linked, being both application events and actions, are specified by the application logic concern (Figure 4.1 number 4). Note that the queries triggered by application events are once more specified in the connection logic concern, which is discussed further on.

Application Actions: UI calls the Application

UI events will trigger the application when certain application behaviour needs to be executed or when the UI requires certain application information or data. For example in the calculator, clicking the equals button results in the application computing the result. For updating the display component, the calculator interface will query the application for its calculated result. Note that there is not necessarily a one-to-one mapping between the value of a component and a data value in the application. For instance, if the calculator UI provides a decimal interface to a binary calculator application, the value of the display component is a transformation of the application's binary result into a decimal value.

Calling the application is achieved by sending a message to the underlying application. As DEUCE relies on SOUL's symbiosis with Smalltalk, this boils down to sending a Smalltalk message to the application that is associated with the UI. For instance in the code snippet below, the rule on lines 1–2 sends the `compute` message to the application and triggers the computation behaviour in the application. The `result` rule on lines 3–4 is called to inquire the application for the value of the calculation. The result of this inquiry is bound to the `?value` variable with the `equals` predicate.

```
1  computeResult(+?appl) if
2     [?appl compute. true]
```

```
result(+?appl,-?value) if
   equals(?value,[?appl result])
```

These application action rules will be triggered upon clicking certain buttons in the user interface. For instance clicking the equals button triggers the `computeResult` rule, as will be explained in Section 4.3.

Application Events: Application calls the UI

When application behaviour is called programmatically, the user interface might not be apprised of a change for which it is required to be updated. Hence the application is responsible for informing the user interface. These connections are also to be specified by the programmer in the application logic concern, and are what we call application events. For instance in the calculator, changing the calculation result in the application should trigger the UI (and hence call DEUCE). The programmer specifies this as follows

```
1  applicationEvent(resultChanged,calculator,#result:).
2  role(calculator,[Deuce.Calculator])

3  linkApplicationEventToQuery(resultChanged,updateResult(?ui)).
```

The `applicationEvent` fact on line 1 expresses that the application event `resultChanged` is called when the `result:` message is sent to the application class that plays the role of the `calculator`. Just as with UI events, this statement de-couples the application event from the actual method and class in the underlying application because one application event can be linked with several methods from the underlying application. Linking the role used in the application event with the corresponding class is expressed by the fact on line 2. In different applications this role can be fulfilled by different classes. If so, this role fact will change. As a result, upon calling the `result:` method in the class `Deuce.Calculator`, DEUCE will be launched with the query `updateResult` (line 3). The query itself is specified in the connection logic (see Section 4.3) and might trigger UI actions. Hence, the application is linked, through application events, with the user interface.

4.3.3 Connection Logic

The presentation logic is concerned with expressing user interface actions and the application logic with expressing application actions. The connection logic brings the two together (Figure 4.1 number 5). After all, logic gets triggered upon events in either UI or application. DEUCE installs the necessary mechanisms (see Chapter 5) behind the scenes such that for both UI events and application events, DEUCE is launched and a corresponding query is executed. These queries are part of the connection logic specification and can trigger other queries in either or both the presentation logic and application logic.

As an illustration, the code example below shows an extract of the connection logic for the calculator. Upon clicking the equals button (i.e. event in the UI), the application computes the result (i.e. application action) and the UI applies the button states (i.e. UI action) corresponding to this event, for instance disabling the equals button for the standard calculator.

```
1  operatorClicked(equals,?ui) if
2     application(?ui,?appl),
3     computeResult(?appl,?result),
4     states(equals,?ui),
```

The `operatorClicked` rule on line 1 for the equals button looks up the application instance corresponding to the current user interface instance, which is bound to the `?ui` variable (line 2). A call is made to the presentation logic by triggering the UI action predicate `states` (line 4) and the application logic is called by triggering the application action predicate `computeResult` (line 3).

Similarly, upon changing the result in the application (i.e. event in the application), the UI will update the display component (i.e. UI action). Recall that the application action `resultChanged` was linked with the `updateResult` query in the application logic concern.

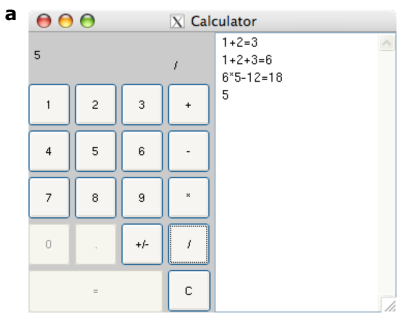
```
1  updateResult(?ui) if
2     application(?ui,?appl),
3     result(?appl,?result),
4     showResult(?ui,?result)
```

The `updateResult` query looks up the application instance corresponding with the current user interface instance, bound to the `?ui` variable (line 2). Next it triggers the application logic to retrieve the result value (line 3) and the presentation logic to show this result (line 4).

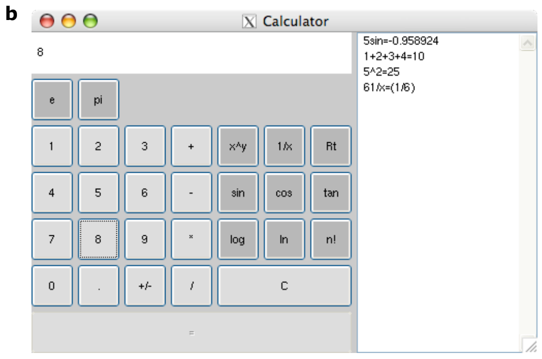
4.4 Revisiting the Calculator Application

In the previous section we have shown how to implement an application with DEUCE. In this section we show how DEUCE is used to evolve from the standard calculator in Figure 3.1 into the calculators in Figure 4.3. Doing so will give a better feeling of how DEUCE is used in practice, and more particularly when evolving a user interface.

The standard calculator is evolved into an extended version by adding a ‘paper tape’ component that will keep a log of previous calculations, as is shown in Figure 4.3a). Next it is shown how a scientific version of the calculator (Figure 4.3b) extends the standard version with extra operators and special numbers. The VisualWorks Smalltalk Painter tool is used to add the new components to the component ‘database’ for the UI by placing them on a canvas as is shown in Figure 4.4. Remember that it is not necessary to carefully adjust the components’ positions.



```
1+2=3
1+2+3=6
6*5-12=18
5
```



```
5sin=-0.958924
1+2+3+4=10
5^2=25
61/x=(1/6)
```

Figure 4.3: Two modes for a calculator: a) standard (extended) b) scientific

4.4.1 Extending the Standard Calculator

The standard calculator is extended with a paperTape which will keep a log of previous calculations. Although this is a small addition to the standard calculator, it already has its effect on the UI concerns. This new paperTape component does not require new behaviour from the application's side and therefore it should not affect the application code. It does however require changes throughout the several UI concerns. The visualisation concern is updated to incorporate the new component. The UI behaviour is changed because different UI actions will now also require the paperTape to be updated.

Visualisation The calculator visualisation needs to be changed such that the new component becomes part of the interface. This is done by adding a `containsComponent` fact for adding the paperTape. The previous alternatives for `containsComponent` remain valid.

```
containsComponent(standardCalculator,paperTape>)
```

Note that these rules can easily be generated by a tool in which the developer selects all the components that are to be part of the interface.

Additionally, adding the paperTape to the standard calculator requires the following extra layout rules.

```
1 leftOf(operators,paperTape).

2 bottomAlign(paperTape,equals).
3 topAlign(paperTape,display)
```

These rules position the paperTape at the right side of the calculator (line 1) and adjust it's height to align with the top and bottom of the other components (lines 2– 3).

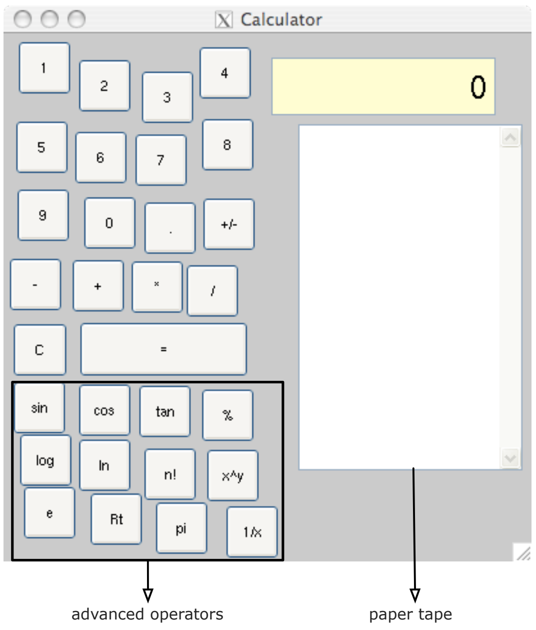


Figure 4.4: Components in the extended calculator example

Behaviour The function of the paper tape is to keep a log of previous calculations by adding operands, operators and computed results to the log. It operates differently from the display component since the display component reflects every change (digit or operator click), while the paper tape only shows the actually computed operations. For instance, when inputting the operator and clicking different buttons, each click is reflected by showing the corresponding operator in the display. However, only when the operator is actually chosen and set, is this operator shown on the log. Intermediate results are not reflected on the paper tape, but sequential calculations are, as is shown in Figure 4.3.

Each time an operand or operator is set or the equals button is called, the `paperTape` component is updated. The UI state for each of these components is changed. For example, when the UI enters the 'equals clicked' state, the previously defined actions remain valid, and thus the previous state rules apply. In addition the paper tape is updated, which is specified by an additional state rule:

```
states(equals,?ui) if
   application(?ui,?appl),
   result(?appl,?result),
   updatePaperTape(?ui,equals),
   updatePaperTape(?ui,?result),
   updatePaperTape(?ui,newline)
```

This rule queries the application for the latest calculated result, and displays the equals sign (line 4), the result (line 5) and a newline (line 6).

Similarly, a new states rule is added for when an operator or operand is set:

```
operatorSet(?ui) if
   getOperator(?ui,?operator),
   displayValue(?operator,?text),
   updatePaperTape(?ui,?text).

operandSet(?ui) if
   getOperand(?ui,?op),
   updatePaperTape(?ui,?op)
```

Recall that `displayValue` retrieves a value to show in a display field such that for instance the `plus` operator is represented by `+`.

The `updatePaperTape` query is also part of the presentation logic as it concerns changing the property of a UI component. It uses the `putToPaperTape` query which retrieves the `paperTape` component from the UI and adds text to it.

```
updatePaperTape(?ui,?value) if
   putToPaperTape(?ui,?value).

putToPaperTape(?ui,?value) if
   addText(paperTape,?ui,?value)
```

For the equals sign and newline putToPaperTape is called with the following values:

```
updatePaperTape(?ui,equals) if
   putToPaperTape(?ui,['=']),!.

updatePaperTape(?ui,newline) if
   putToPaperTape(?ui,[String with: Character cr]),!
```

There are no UI events defined on the new log component, and therefore no new links are made between UI events and queries.

Note that the paperTape component is updated when entering existing UI states. The UI state does not change but does require additional UI actions to be triggered. Also no additional links are made between the application and the UI as the paper tape does not trigger any functionality of the application or vice versa. Hence, the application logic concern remains unchanged. The connection logic concern does not change either as the paper tape is updated as a side effect of existing UI states. Because of the declarative nature of the specification, new side effects are added by new rules for the existing state, instead of doing so by changing actual application statements.

4.4.2 A Scientific Calculator

The scientific calculator extends the standard calculator with new operators and special number values. This requires both the application and the UI to evolve. The application has to provide the new behaviour (types of calculations) and the UI is updated with new components to access this behaviour. As new links are created between application and UI, the application will evolve, and so will the connection logic and the presentation logic. As we will see in this section, each of the concern evolutions can be thought of in separation of the others.

Presentation Logic

We start with specifying the visualisation logic for the scientific calculator. Recall that the visualisation describes what components are part of the interface, what their properties are, and how they are positioned with respect to one another.

Adding Components The `containsComponent` facts, as shown in the next code snippet, is used to list the components that will be part of the scientific calculator interface. Remember that these components were put on a canvas with the Visualworks Smalltalk UI Painter tool and are assigned a unique name which is used in the logic facts to refer to these components.

```
containsComponent(scientificCalculator,one).
containsComponent(scientificCalculator,two).
containsComponent(scientificCalculator,three).
...
containsComponent(scientificCalculator,equals).
containsComponent(scientificCalculator,plus).
containsComponent(scientificCalculator,minus).
containsComponent(scientificCalculator,multiply).
...
containsComponent(scientificCalculator,sinus).
containsComponent(scientificCalculator,cosinus).
...
containsComponent(scientificCalculator,paperTape)
```

Components can be grouped together, such that these groups are used to specify layout or change properties of the whole group of components at once. The standard calculator already defined groups for the digits and operators. For the scientific calculator the additional groups added are the following:

```
group(advancedOperators,
        <power,reciprocal,root,sinus,cosinus,tangens,log,ln,factorial>).
group(specialValues,<e,pi>).
```

Using these group specifications, the `containsComponents` facts can also be expressed as

```
containsComponent(scientificCalculator,digits).
containsComponent(scientificCalculator,specialDigits).
containsComponent(scientificCalculator,operators).
containsComponent(scientificCalculator,advancedOperators).
containsComponent(scientificCalculator,result).
containsComponent(scientificCalculator,display).
containsComponent(scientificCalculator,paperTape).
containsComponent(scientificCalculator,specialValues)
```

Specifying Component Properties By specifying component properties the developer sets the colour, font and size of the UI components. For the standard calculator we defined components to be at least 40 by 40 pixels, to have a grey background colour and a black font text. For some components these properties are made more specific by adding additional rules. For instance in the scientific calculator the advanced operators and special value buttons are given a different colour with:

```
backgroundColour(?componentName,lightGray) if
   group(specialValues,?comps),
   includes(?componentName,?comps).

backgroundColour(?componentName,lightGray) if
   group(advancedOperators,?comps),
   includes(?componentName,?comps)
```

These rules link the colour of a component to the truth of the body of the rule, which specifies that the component should be part of either the group `specialValues` or `advancedOperators`, as was specified by the developer elsewhere in the visualisation logic concern. This rule can be generalised and moved to the DEUCE core logic such that in the future the developer can specify that properties for an entire set of components without having to know about logic rules.

Layout Similar to the standard calculator, the components in the scientific calculator are laid out by a set of layout specifications. For instance, digit buttons and operators are positioned with:

```
fillRows(digits,3).
fillRows(specialDigits,3).

oneColumn(operators).
fillRows(advancedOperators,3).

oneRow(specialValues).
```

The `fillRows` statement spreads components from a group (e.g. digits) over several rows such that each row consists of three components. Now that components within the groups are positioned, we position the groups with respect to one another.

```
above(specialValues,digits).
above(digits,specialDigits).
above(specialDigits,result).

leftOf(digits,operators).
leftOf(operators,advancedOperators).
leftOf(advancedOperators,paperTape).

topAlign(three,plus).
topAlign(plus,power)
```

Note that grouping the components allows us to specify layout relations for the entire group, but this does not necessarily have to be the case. Groups do not need to be regarded as being a visual group and layouts can be specified without regarding group as well.

The display component is positioned on top of the `specialValues` group. One of its subcomponents, namely the display showing the inputted digits, needs to span the entire width from the one button to the root button and hence it is left and right aligned with these components.

```
above(display,specialValues).
leftAlign(numberDisplay,one).
rightAlign(numberDisplay,root)
```

The same is done for the equals component, the clear component and for the paperTape which spans the entire height.

```
leftAlign(equals,display).
rightAlign(equals,factorial).

rightAlign(clear,root).
leftAlign(clear,reciprocal).

topAlign(paperTape,display).
bottomAlign(paperTape,equals)
```

This concludes the additional visualisation for the scientific calculator: additional components are created, their properties are set and the adapted layout is specified. Next, the UI behaviour needs to be specified.

Behaviour: UI actions The UI actions describe how properties of the UI and its components change. The actions as defined in the standard calculator also stand for the scientific calculator, such as the following rules that specify that number input is allowed after clicking an operator button.

```
states(?operator,?ui) if
   digitsAllowed(?ui)
```

For the standard calculator it was also specified that the equals button is disabled unless a second operand was input. However, as operations in the scientific calculator can also be unary operators, two additional rules are added for making a distinction between unary and binary operators.

```
states(?operator,?ui) if
   isUnaryOperator(?operator,?ui),
   firstOperandSet(?ui),
   enable(equals,?ui).

states(?operator,?ui) if
   isBinary(?operator,?ui),
   firstOperandSet(?ui),
   secondOperandSet(?ui),
   enable(equals,?ui)
```

Behaviour: UI events The UI logic can be triggered by UI events (or application events). These UI events are launched by manipulating a component, for example when clicking a button. In the presentation logic concern the developer specifies what events are allowed on what components. For example, the following code snippet specifies that the click event on the sinus and power button, will launch the `operatorClicked` query with certain values. The other operators have a similar specification.

```
UIEvent(sinusClicked,sinus,click).
UIEvent(powerClicked,power,click).

linkUIEventToQuery(sinusClicked,operatorClicked(sinus,?ui)).
linkUIEventToQuery(powerClicked,operatorClicked(power,?ui))
```

Application Logic

The underlying application code for the calculator has to provide for the new functionality of scientific operations. If the Smalltalk calculator cannot compute a sinus function, there is no use in providing it in the user interface. However, recall that if the application needs to evolve, this is done without having to consider the user interface.

The application logic concern, being part of the user interface concerns, specifies the link between the interface and the application. For instance it specifies how operands and operators in the application can be set and calculations can be executed.

Application Actions Application actions are used by the UI concerns to trigger behaviour and to query the underlying application. As the scientific calculator distinguishes between unary and binary operators, the application actions to query the underlying application about what kind of operator is being used, have to be added to this concern.

```
isUnaryOperator(+?appl,?op) if
   [?appl unaryOperator: ?op]

isBinaryOperator(+?appl,?op) if
   [?appl binaryOperator: ?op]
```

Application Events The underlying application calculator does not need to call upon the UI in any additional cases, and therefore no new application events where added to this concern.

Connection Logic

The connection logic links application actions and events with UI actions and events. In the standard calculator specification this concern expresses the rules for the UI queries that are launched, such as digitClicked and operatorClicked and the application queries that are launched, such as updateResult, operandSet and operatorSet. In the scientific calculator these standard calculator specifications are still valid, but the operatorClicked is extended. As the scientific calculator distinguishes between unary and binary operators, there are not always two operands needed for an operator to be executed. Hence, the operator is not only set when clicking a second operand, but also when clicking the equals button. Recall that the button states enable this equals button upon clicking the unary operator.

```
operatorClicked(equals,?ui) if
   application(?ui,?application),
   getOperator(?ui,?operator),
   applicationValue(?operator,?op),
   isUnaryOperator(?application,?op),
   operator(?application,?op)
```

The application logic and the presentation logic are evolved in separation of one another and remain oblivious to the other concern. The connection logic however has to provide the necessary queries to be launched upon UI and application events. These queries bring the two other concerns together. As such it expresses the application behaviour that is reflected by the user interface. However, although it makes use of the specifications of the two other concerns, it does not change them.

4.5 Conclusion

In Chapter 3 we proposed five requirements which a conceptual solution should adhere to in order to provide for a separation of user interface concerns. Two of these requirements stated that UI concerns should be specified in separation of each other and at a high-level of abstraction. With DEUCE we provide an implementation to instantiate this conceptual solution and fulfil these requirements. The UI developers (or designers) will specify the user interface concerns at a high-level. In this chapter we have shown how this is achieved with DEUCE. How these high-level specifications are transformed into an actual Visualworks Smalltalk interface and how the are linked with an application, will be shown in the following chapter. It will discuss DEUCE's core logic and the mechanisms in the DEUCE kernel that combine the core logic with the high-level specification.

Note that in order to create an instantiation of the conceptual solution we have chosen to use SOUL. SOUL is a declarative meta programming language on top of Visualworks Smalltalk. As such it provides a language to express the UI concerns declaratively as well as an underlying technology to transform this declarations into actual application level objects. SOUL is written in symbiosis with Smalltalk, which is why DEUCE currently is applied for Visualworks Smalltalk applications and creates Visualworks Smalltalk user interfaces. Nevertheless SOUL is currently also being used to reason upon Java code and can be used to generate text or code [Bri07]. This will allow DEUCE to create non-Smalltalk UIs in future work.

Chapter 5

The Internals of DEUCE

In Chapter 3 we introduced requirements which an ideal solution should adhere to in order to achieve a true separation of concerns. In Chapter 4 we motivated the choice for a declarative approach for implementing an instantiation of such a solution and we showed how developers use DEUCE for specifying a high-level user interface. Behind the scenes DEUCE will transform these high-level UI specifications into an actual low-level running application. In this chapter we provide the details of DEUCE's internal mechanisms that are used to implement our solution for a separation of user interface concerns. We start by giving a global overview on the DEUCE architecture in Section 5.1. Section 5.2 explains what a final application looks like in DEUCE, before the following sections 5.3–5.8 show how to create it. In Section 5.9 we summarise how the requirements proposed in Chapter 3 are met and we give a critical view on the provided implementation.

5.1 Overview of the DEUCE Architecture

As elaborated on in Chapter 4, DEUCE uses a declarative approach for supporting developers in separating user interface concerns. As a declarative language we have chosen SOUL (see Section 4.2) which is implemented on top of Visualworks Smalltalk. Hence, the implementation provided by DEUCE, will be used to create Visualworks Smalltalk systems. The developers implement the application code in Smalltalk and specify the user interface at the logic level, while DEUCE composes the actual runtime system. The DEUCE architecture is depicted in Figure 5.1 and distinguishes between three levels. The logic level contains the high-level UI specification created by the developers as illustrated in Chapter 4 and a set of rules to reason upon this specification and transform it into a runtime system. This runtime system resides at the application level and consists of an application and its user interface. The level in-between, being the Smalltalk level, contains components to actually build a Smalltalk system and provides the mechanisms for the interaction between application and UI. Before we discuss each of the components in the DEUCE architecture in detail in the remainder of this chapter, we give an overview with respect to the UI concerns we distinguished in Chapter 3.

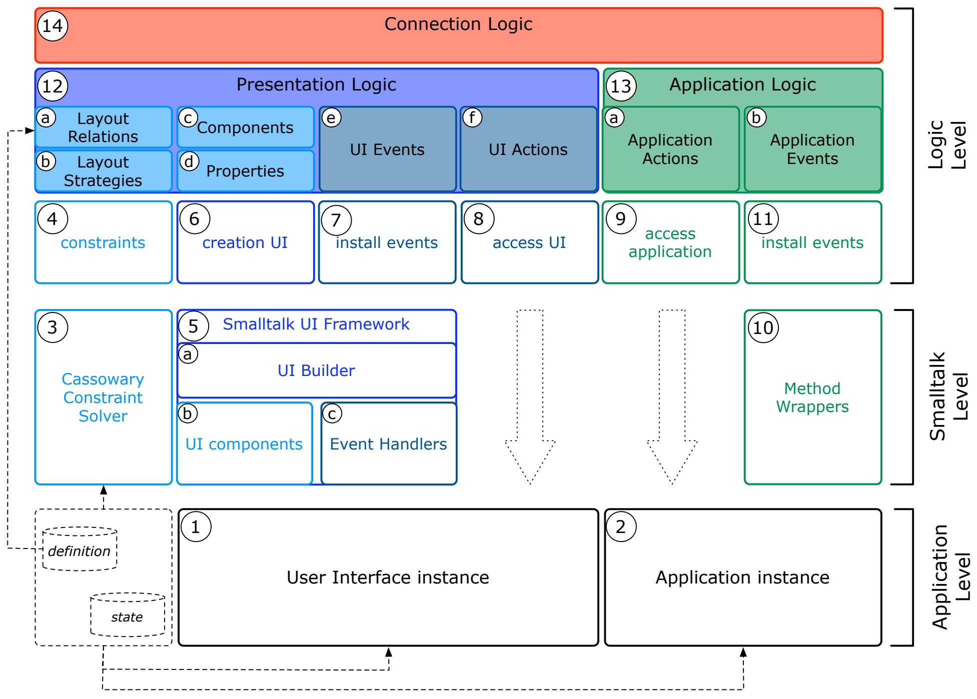


Figure 5.1: DEUCE architecture

Running System

Developers use DEUCE to specify the user interface concerns at a high level in separation of one another. The underlying application itself is written in Visualworks Smalltalk. Together these are transformed into the final running system. This system consists of an application instance and a corresponding user interface instance. The user interface instance (Figure 5.1 number 1) is a Visualworks Smalltalk UI generated by DEUCE and contains UI components. Each component has its specified properties set and event handlers installed. The application instance (number 2) is an instance of the Smalltalk application instrumented with the necessary code to link it to the UI. Installing the event handlers and instrumenting the application code is done behind the scenes by DEUCE and no longer of concern to the developers.

Presentation Logic

Presentation logic (Figure 5.1 number 12) consists of a visualisation and a behaviour part. The visualisation is concerned with what components are part of the UI (number 12c), their properties like colour, font, layout, enablement, and how they are positioned with respect to one another (number 12a). The behaviour specifies what UI events (number 12e) are triggered by the UI (e.g. a button is clicked, a window is closed, a text-field is filled out) and what UI actions (number 12f) are a possible consequence (e.g. a component is enabled, layout positions change, components are added).

The layout specification is transformed into a set of constraints (Figure 5.1 number 4) which are solved by a constraint solver (number 3). The latter is an implementation in Smalltalk of the Cassowary linear constraint solver [Bad01]. Based on the UI component specifications (and their properties), a UI instance (number 6) is constructed by using the Visualworks Smalltalk UI framework (number 5). For each component specification, a Smalltalk UI component is created (number 5b) and the Visualworks Smalltalk UIBuilder (number 5a) creates the actual UI instance which these components are added to. UI events are dealt with by using the Visualworks Smalltalk event handling mechanism (number 5c). DEUCE transforms the UI event specifications into event handlers for the different UI components (number 7). Finally, UI actions correspond to accessing (number 8) the actual UI by sending a Smalltalk message to the UI instance.

Application Logic

Application logic (Figure 5.1 number 13) is used to specify how the application is linked to the UI. On the one hand application actions (number 13a) trigger application behaviour from within the UI. This corresponds to accessing (number 9) the application instance by sending a corresponding Smalltalk message. On the other hand application events (number 13b) are occurrences in the application instance that will trigger the UI. To deal with these events, DEUCE instruments the application code to call the UI (number 11). As a mechanism to provide this code instrumentation, method wrappers are used (number 10).

Connection Logic

The connection logic (Figure 5.1 number 14) brings presentation and application logic together. As mentioned before, both UI and application events can trigger user interface behaviour and application behaviour (actions). Hence, upon an event a connection query will be triggered. A connection query will launch UI actions in the presentation logic and application actions in the application logic. The code behind the Smalltalk event handlers and the Smalltalk method wrappers is responsible for calling DEUCE and launching a connection query. This code is installed when setting up these mechanisms in the presentation and application logic concern.

5.2 Different Parts of the Running Software System

When using DEUCE, a programmer creates an application and a declarative user interface specification. Both are brought together by the mechanisms behind DEUCE to create a running system. The several parts within this running system are depicted in Figure 5.2. It consists of an application instance (Figure 5.1 number 2) and a UI instance (number 1). The running system maintains a link to the original UI specification, the user interface's state and the several windows that are part of the UI, with

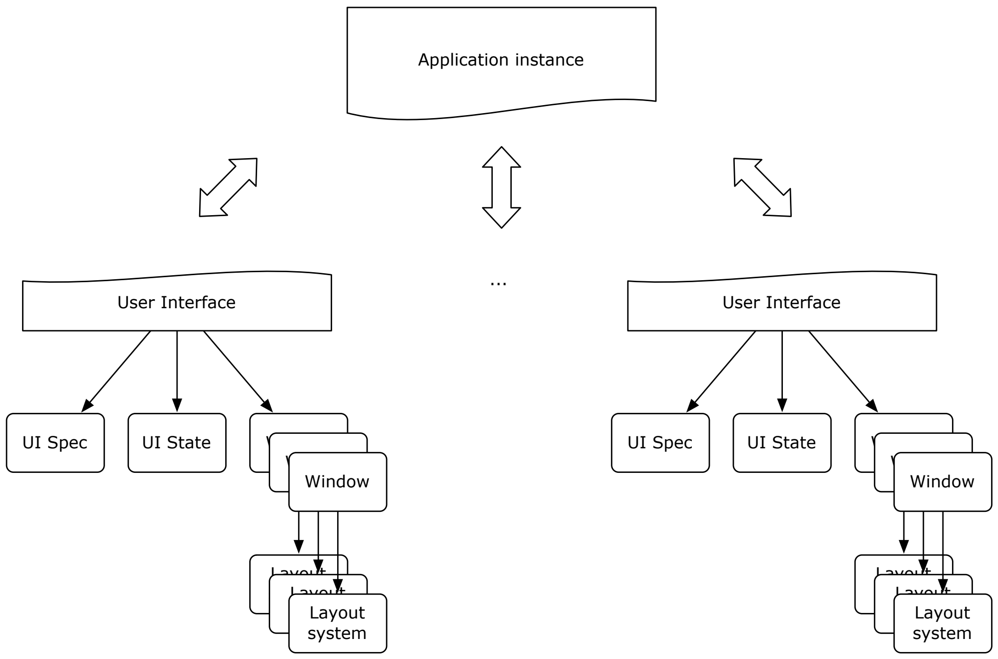


Figure 5.2: The different parts of the running system

each their own layout system. Note that multiple user interfaces can be defined for one application.

Application Instance

The underlying application is developed in separation of the UI. For the DEUCE architecture this means that the application is implemented in Visualworks Smalltalk. An instance of this application will be part of the running system, depicted in Figure 5.1 as number 2. DEUCE will instrument this application instance with code that provides the actual link to the UI. As a matter of fact, application instances are linked back to the DEUCE logic level and the logic level provides the link to the corresponding UI instance. The mechanism to actually link the application to DEUCE, is explained in Section 5.7.

User Interface Instance

DEUCE will create a Visualworks Smalltalk UI (5.1 number 1), based on the high-level UI specification. A user interface has a link to its original specification and the application it is an interface for. It can contain several windows and keeps track of its state.

User Interface Windows

Each of the windows in a user interface is an instance of the Smalltalk `UIBuilder` class. It is responsible for keeping track of the components, display them on the screen, update their properties, and deal with UI events. How a window is created is explained in Section 5.4.

User Interface Specification

The high-level UI specification remains accessible to the runtime application as it contains valuable information which the application might need to respond to at runtime. For instance, contexts change at runtime and a new context might require new components, layout and behaviour to apply. A UI specification, stored in the UISpec repository, is valid for all user interfaces of that kind. Hence, adding rules to the layers of the UISpec will have an influence on all UI instances of that kind.

User Interface State and Layout Solving System

Although a UI specification is shared among all UIs of a certain kind, some knowledge is particular for an instance or one of its windows, such as the state and the layout solving mechanism.

The runtime state of the UI has an influence on its behaviour and appearance, and therefore on the DEUCE reasoning process related to this UI. Furthermore a UI instance might need particular UI specifications that are only valid for that instance. This is for example the case when a certain user wants to apply his own personalised layout guidelines. To keep track of this information, DEUCE creates a state repository for each UI instance.

Also the layout of one window can defer from another window of the same kind. For example it defers because it has more ore less components or a different order for the components. Hence a window has its particular constraint solver which contains the constraint system for that particular window's layout.

Setting up the Runtime System

The logic of DEUCE is modularised into several repositories. Recall from Chapter 4 that triggering a predicate that is part of another repository, is done through repository variables (denoted by `?repositoryVariable->`). In rule examples in the remainder of this chapter the following variables are used to denote the following repositories:

- `?UISpec`: contains the high-level specification as developed in Chapter 4.
- `?storage`: used to retrieve the application instance, state repository and windows for each UI instance, and layout system for each window.
- `?layout`: repository with rules to provide for the automated layout system. In DEUCE this refers to rules to translate layout relations into constraints, to create

a constraint system and constraint variables, and to add and remove variables and constraints to a constraint system.

- `?UISpecRules`: repository that provides the core logic of DEUCE to map the high-level UI specification into the low level platform specific implementation.

The following code snippet shows rules set up the runtime system with DEUCE. The different components of the runtime system which were described above, remain accessible from within the logical level through the storage repository.

```
1  createUI(?appl,?ui,?interface,?definitionRepository) if
2          ?UISpecRules->newUserInterface(?ui),
3          newRepository(?stateRepository),
4          ?storage->add(userInterface,?ui),
5          ?storage->definition(?ui,?definitionRepository),
6          ?storage->state(?ui,?stateRepository),
7          ?storage->application(?ui,?appl),
8          linkWithApplication(?ui),
9          createWindows(?ui),
10         defaultWindow(?ui,?interface)

11 createWindow(?ui,?interface,?inst) if
12         currentUserInterfaceInstance(?ui),
13         ?UISpecRules->newWindow(?interface,?ui,?inst),
14         ?layout->setupLayoutSystem(?inst,?layoutSystem),
15         ?storage->window(?ui,?inst,?interface),
16         ?storage->layoutSystem(?inst,?layoutSystem),
17         ?UISpecRules->addComponentsFromTo(?interface,?inst),
18         ?UISpecRules->all(setWindowProperties(?interface,?inst))
```

To start this process, the arguments `?appl` and `?definitionRepository` should be bound to the application and the UI specification repository respectively. `?interface` represents the name of the window that is used as the default window for this particular user interface. An initial UI instance is created (line 2), along with an empty repository to keep track of the UI's state (line 3). They are stored, together with the application and specification repository links, through the storage repository (lines 4–7). The necessary mechanisms to link the application back to the user interface specification are set up (line 8). The windows that are used in the user interface are created (line 9) and the window with as name `?interface` will serve as the default window (line 10).

How to create a window for a user interface is expressed on lines 11–18. An empty window is generated (line 13) and its constraint system is set up (line 14). Both are stored through the `?storage` repository (lines 15–16). The components that are part of the `?interface` window specification are added to the window instance (line 17). At that point each component will be linked to the application. Finally the initial properties of the window are applied (line 18).

In what follows we give details on how the Visualworks Smalltalk UI is created, components are added to it and how UI and application are linked with each other. We start by explaining the mechanism used in DEUCE to achieve automated layout.

5.3 Automated Layout through Constraint Solving

Layout is an important aspect of a UI's presentation and in order to lift the developers' view and comprehension away from the low-level system, it should be applied automatically. Although some UI builders provide design policies, and machine learning techniques allow to determine the layout upon usage of the interface, a vast majority of systems that provide automated layout use a constraint-based method [Lok01]. With a constraint system the developers still provide the layout (as opposed to machine learning techniques), but have the power to construct more complex hierarchical layouts (as opposed to design policies).

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.

Bados et al. [Bad01] investigated what properties are needed for a constraint solver when used for user interfaces. First of all, both equality and inequality constraints are needed. For instance, when specifying that a component A should be positioned to the left of a component B, this means that A.rightside $\leq$ B.leftside. Secondly, when it should be possible to express that component A should always remain fixed while component B can be repositioned, the constraint fixing A to a position should have preference over (i.e. be stronger than) the constraint positioning B to the right of A. Hence, the constraint system has to allow for specifying preferences on constraints. Thirdly, the solver needs to cope with cycles as it is hard to avoid them and undesirable to leave the responsibility of avoiding cycles to the developers. A constraint solver that allows cycles, allows the same variable to be used in different constraints.

In addition, it is required for the constraint solver to solve its system incrementally. This allows for faster recalculation of positions without having to resolve the whole system, which would be the case if the solver is not incremental. Since layout deals with coordinates of UI components, the domain upon which the solver operates is numerical and discrete. Therefore a numerical constraint solver is advisable, as opposed to symbolic solvers. Although using another type of constraint solver does not affect the conceptual solution of DEUCE, we have chosen to use a linear arithmetic constraint solving algorithm as proposed by Badros et al. [Bad01], namely the Cassowary algorithm. This algorithm has proven to be valid within the context of automated layout for UIs. The high-level specification (Figure 5.1 number 12a) is transformed (Figure 5.1 number

4) into a constraint system, which will be solved by a Cassowary constraint solver (Figure 5.1 number 3). Next the Cassowary constraint solver is explained. Afterwards we show how this solver is used by DEUCE to achieve automated layout.

5.3.1 The Cassowary Constraint Solver

DEUCE uses a Smalltalk implementation of Cassowary (Figure 5.1 number 3). Cassowary's algorithm and implementation is thoroughly described in [Bad01]. We will only highlight how to use it from the perspective of a user using Cassowary.

To set up a constraint system, the programmer defines constraint variables and constraints, and adds them to the constraint solver. Constraint variables have a `name` and `value`. The latter will hold the value of the variable after the constraint system has been solved. In analogy to numbers, constraint variables understand the operations @ (to create points), $+$, $-$, $*$, $/$, $=$ (`cnEqual`), $\geq$ (`cnGEQ`), $\leq$ (`cnLEQ`). The last three operators can take a strength as an extra argument such that some constraints can be given a stronger preference over others. Applying these operators results in a linear constraint.

Linear constraints have a linear equality or inequality expression between constraint variables and are typically used for describing structures such as geometric layouts. *Stay and edit constraints* are specified for a single constraint variable, and give hints for sizing and positioning the layouts [Hos01]. Edit constraints are used to update a variable value and typically facilitate operations for moving objects. Stay constraints try to fix a variable value and preserve its previous value if possible. Linear constraints are usually strong, whereas stay and edit constraints are weak. Cassowary allows to label constraints with strengths such as strong, medium, and weak.

An additional type of constraint is the bounds constraint which can be used to add a lower and upper bound to a variable. Constraints can be added and removed from the constraint solver. When in `autoSolve` mode, the constraint solver will automatically search for a solution each time a constraint is added or removed. Otherwise the solver needs to be started manually.

5.3.2 Basic Layout Relations as Cassowary Constraints

As the Cassowary constraint solver is used to achieve an automated layout in DEUCE, the high-level layout specification is transformed into an instance of the Cassowary constraint solver implementation in Smalltalk. DEUCE provides a set of rules (Figure 5.1 number 4) at the logic level for creating and solving this constraint solver instance. These rules are used internally only and developers using DEUCE do not have to deal with these. Both instantiating a constraint system as well as solving it, is managed from within DEUCE by sending messages to the Cassowary implementation.

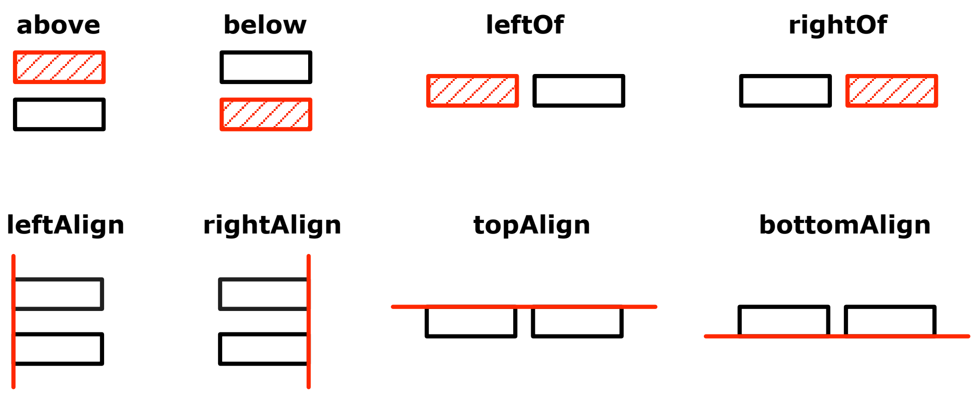


Figure 5.3: Basic positioning relationships between components

Basic Layout Specifications

The basic positioning relations we have adopted are depicted in Figure 5.3. The four relations to position components adjacent to one another are above, below, left-of, and right-of. The basic relations to align components with respect to one-another are top-align, bottom-align, left-align, and-right align.

A UI component layout can be described by means of its right, left, top, bottom and centre points. Based on these points, the basic layout relations can be translated into linear equations, as shown in table 5.1. For example, component a is positioned above component b if its bottom equals or is greater than the top of component b.

Table 5.1: Layout relations for positioning user interface components

layout relation	Linear Equation
a above b	$bottom \quad a \leq top \quad b$
a below b	$top \quad a \leq bottom \quad b$
a left-of b	$right \quad a \leq left \quad b$
a right-of b	$left \quad a \leq right \quad b$
align top of a and b	$top \quad a = top \quad b$
align bottom of a and b	$bottom \quad a = bottom \quad b$
align left of a and b	$left \quad a = left \quad b$
align right of a and b	$right \quad a = right \quad b$

In DEUCE, these basic layout relations are transformed into constraint relations. For each linear equation the constraint variables are created (or looked up if already part of the constraint solver) and the corresponding constraint is added to the solver. Consider the following code snippet for transforming the layout relation `above(?x,?y)`.

```
1  adjoin(above,?comp1,?comp2,?solver) if
2    constraintVariable(?comp1,bottom,?comp1var,?solver),
3    constraintVariable(?comp2,top,?comp2var,?solver),
4    makeConstraint(greater,?comp2var,?comp1var,strong,?solver)
```

A constraint variable is retrieved for the `?compA` (line 2) and `?compB` (line 3) components and a greater-than relation is created between the two (line 4). A constraint is added to the constraint solver by the rule

```
1  makeConstraint(greater,?xeq,?yeq,?strength,?solver) if
2    [?solver addConstraint:
3          (?xeq cnGEQ: ?yeq
4          strength: (Cassowary.ClStrength perform: ?strength)). true]
```

This rule translates the greater-than relation into the corresponding Cassowary constraint (line 3–4) and adds it to the constraint solver `?solver` (line 2).

Sizing Relations

Another set of layout relations is related to the size of the components. A minimum or fixed height and width, as well as the boundaries of the display window are also enforced by the constraint system. Table 5.2 shows how these relations correspond to linear equations. For example, if the minimum height of component `a` should be `d` this means that the bottom of component `a` equals or is greater than its top plus `d`.

Table 5.2: Layout relations for sizing user interface components

layout relation	Linear Equation
minimum height of a is d	$(top\quad a \quad + \quad d) \leq bottom \quad a$
minimum width of a is d	$(left\quad a \quad + \quad d) \leq right \quad a$
fixed height for a is d	$(top\quad a \quad + \quad d) = bottom \quad a$
fixed width for a is d	$(left\quad a \quad + \quad d) = right \quad a$

In DEUCE, again a corresponding constraint relation is created and added to the solver. For instance constraining the minimum height of a component is done with

```
1  size(height,minimum,?compA,?min,?solver) if
2     constraintVariable(?compA,top,?topVar,?solver),
3     constraintVariable(?compA,bottom,?bottomVar,?solver),
4     makeEquation(sum,?topVar,?min,?xeq),
5     makeConstraint(greater,?bottomVar,?xeq,required,?solver)
```

The corresponding constraint variables for the top (line 2) and bottom (line 3) of the `?compA` component are retrieved. The left side of the equation is constructed (line 4) and the constraint is added to the constraint solver `?solver` (line 5).

To specify that a component should remain within a window's bounds, the bounding box constraint can be used. DEUCE transforms this relation into a bounds constraint as shown below.

```
withinBoundsOf(?comp,?boundingBox,?solver) if
   constraintVariable(?comp,top,?topVar,?solver),
   constraintVariable(?comp,left,?leftVar,?solver),
   withinBounds(?topVar,[?boundingBox origin x],
                       [?boundingBox origin y],?solver),
   withinBounds(?leftVar,[?boundingBox corner x],
                       [?boundingBox corner y],?solver).

withinBounds(?var,?lower,?upper,?solver) if
   [?solver addBounds: ?var lowerBound: ?lower
                             upperBound: ?upper. true]
```

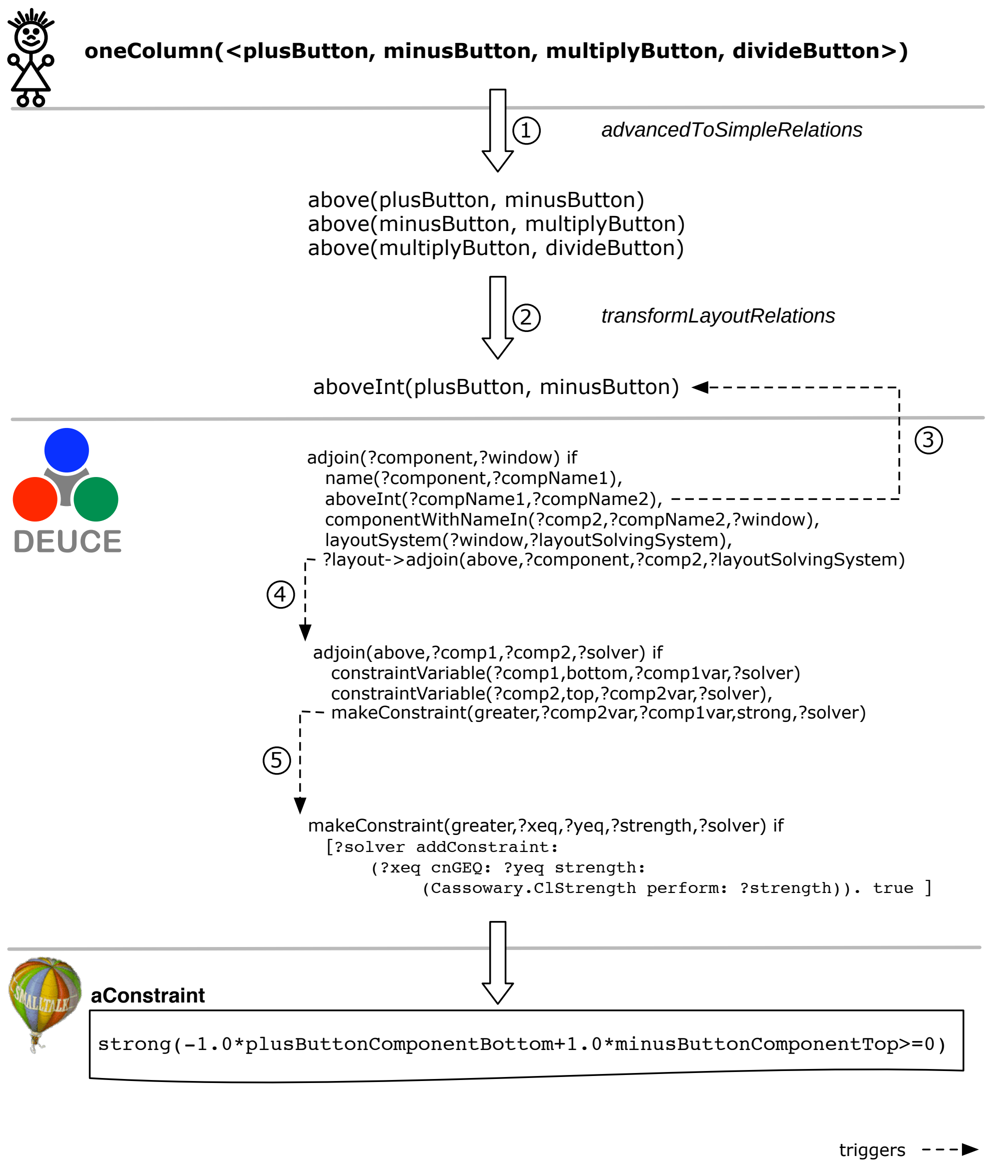
Constraint Strengths

Constraint strengths can be used to specify preferred and required layout properties. For adjacency and alignment constraints, the constraints are specified to be of *strong* strength. Preferably these constraints have to be satisfied, but they can be deviated from if necessary. The sizing constraints have a *required* strength as components need to stick to their minimum or fixed size. The same is true for the bounds constraints as it is necessary for all components to stay within the boundaries of the UI window.

5.3.3 From High-Level Layout Specifications to Basic Relations

Part of the DEUCE core logic contains the rules to transform the high-level layout specification (Figure 5.1 number 12a) to the rules described above, which will add the actual constraint to the constraint solver instance. This process is illustrated in Figure 5.4. Remember that this process is part of DEUCE internal mechanism and that only the high-level specification is of concern to the UI developers. Nevertheless, for now the developers have to make sure that two constraints are not conflicting and future support will need to deal with this.

First all advanced layout relations (e.g. oneColumn) are translated to basic relations (Figure 5.4 number 1), which in their turn are translated to internal basic layout relations (Figure 5.4 number 2). These basic relations are expressed in terms of single UI components only. The reason for this transformation is to make sure all layout relations, also the ones expressed in-between groups, are considered. The internal relations (Figure 5.4 number 3) are translated into constraints (Figure 5.4 number 4) and added to the constraint solver (Figure 5.4 number 5).


oneColumn(<plusButton, minusButton, multiplyButton, divideButton>)
1
advancedToSimpleRelations
above(plusButton, minusButton)
above(minusButton, multiplyButton)
above(multiplyButton, divideButton)
2
transformLayoutRelations
aboveInt(plusButton, minusButton)
3
DEUCE
4
5
aConstraint
triggers

```
adjoin(?component,?window) if
   name(?component,?compName1),
   aboveInt(?compName1,?compName2),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?layoutSolvingSystem),
   ?layout->adjoin(above,?component,?comp2,?layoutSolvingSystem)
```

```
adjoin(above,?comp1,?comp2,?solver) if
   constraintVariable(?comp1,bottom,?comp1var,?solver)
   constraintVariable(?comp2,top,?comp2var,?solver),
   makeConstraint(greater,?comp2var,?comp1var,strong,?solver)
```

```
makeConstraint(greater,?xeq,?yeq,?strength,?solver) if
  [?solver addConstraint:
       (?xeq cnGEQ: ?yeq strength:
            (Cassowary.ClStrength perform: ?strength)). true ]
```

```
strong(-1.0*plusButtonComponentBottom+1.0*minusButtonComponentTop>=0)
```

Figure 5.4: From high-level layout specification to low-level constraint

Transforming into Basic Relations for Single UI Components

Basic layout relations can be expressed between components, but also between sets of components and groups. If one group is positioned above another group, this implies that all components within the first group are positioned above all components within the second group. Repositioning a component in the first group might have an effect on all components in the second group. In order for these constraints to apply, DEUCE transforms all basic relations between components, sets of components and groups into a set of internal basic relations which apply to single UI components only (Figure 5.4 number 2). For example the following rule asserts internal above relations:

```
transformAbove(?comp1,?list2) if
    findall(?comp1, and(includes(?comp2,?list2),
            addSpecification(aboveInt(?comp1,?comp2))),?result)
```

This rule is called with `?comp1` bound to a UI component and `?list2` bound to a list of single UI components. It is called by other rules that will break down groups/lists of groups/lists of components into a list of single components. Also groups are expanded into groups of single UI components and added to the UI specification, with:

```
expandGroups if
   expandGroup(?group,?components),
   addSpecification(groupInt(?group,?components)).

expandGroup(?group,?components) if
   ?UISpec->group(?group,?comps),
   findall(?sublist,and(includes(?comp,?comps),
                     expandGroup(?comp,?sublist)),?sublists),
   flatten(?sublists,?components)
```

Basic Relations to Constraints

Now that the basic relations are expressed in terms of single UI components, they can be added to a constraint system. As seen before both basic positioning and alignment as sizing relations are expressed with constraints. For instance, the above relation is added to the constraint solver by the following code snippet:

```
1  adjoin(?component,?window) if
2     name(?component,?compName1),
3     aboveInt(?compName1,?compName2),
4     componentWithNameIn(?comp2,?compName2,?window),
5     layoutSystem(?window,?layoutSolvingSystem),
6     ?layout->adjoin(above,?component,?comp2,?layoutSolvingSystem)
```

For each internal `above` relationship (line 3) that is specified in the UI specification (Figure 5.4 number 3) for a certain component (line 2), the other components in the relationship are retrieved from the UI instance (line 4). The layout system (here a constraint solver) for the window is looked up (line 5) and the relation is transformed into a constraint which is added to that solver (line 6), as is explained in Section 5.3.2 (Figure 5.4 number 4 and 5).

Also sizing and alignment relations are transformed into constraints. For example, the following adds a constraint to give components a minimum height.

```
minimumHeight(?component,?ui) if
   name(?component,?compName1),
   ?UISpec->minimumHeight(?compName1,?min),
   layoutSystem(?ui,?system),
   ?layout->size(height,minimum,?component,?min,?system)
```

Advanced Layout Relations

Based on the basic layout relations, more advanced layout relations can be expressed (Figure 5.4 number 1). For instance, putting a set of components in one row signifies that all components are put to the left of the component next to it. Transforming a `oneRow` specification into `leftOf` specifications, is done by the DEUCE rule in the following code example. The derived `leftOf` facts are added to the UI specification with the `addSpecification` predicate. The first rule applies for two components whereas the second rule goes through a list of components.

```
oneRowToLeftOf(<?firstComponent,?secondComponent>) if
   addSpecification(leftOf(?firstComponent,?secondComponent))

oneRowToLeftOf(<?firstComponent|?restComponents>) if
   head(?secondComponent,?restComponents),
   addSpecification(leftOf(?firstComponent,?secondComponent)),
   oneRowToLeftOf(?restComponents)
```

One can continue building other relations on top of this as well. For instance the following illustrates how putting components in a diagonal boils down to putting every component above and to the left of every other component, or putting the components both in one row and one column.

```
diagonalToColumnAndRow(?comps) if
   addSpecification(oneRow(?comps)),
   addSpecification(oneColumn(?comps))
```

The developers are no longer concerned with the actual layout position of components and can therefore reason about their position in more abstract terms. This introduction of high-level abstractions will increase the evolvability of UIs.

5.4 Creating the Visualworks Smalltalk UI

DEUCE creates a running Visualworks Smalltalk UI based on the high-level UI specification. The high-level specifications are translated through a set or rules (Figure 5.1 number 6) into actual Smalltalk objects of which a Visualworks Smalltalk UI is built up (Figure 5.1 number 5). In this section we introduce how UIs are typically built with the Visualworks Smalltalk UI Framework. As some of these steps require a manual intervention from the developers, we next show how these steps can be omitted and how a UI can be created programmatically. The latter is used by DEUCE for transforming the high-level specification into an running Smalltalk UI.

5.4.1 The Visualworks Smalltalk `windowSpec`

When creating a user interface in Visualworks Smalltalk, developers use the UI Painter Tool to create the UI by laying out its static graphical visualisation on a canvas. The Painter Tool is used to specify the properties for each of the components. These include positioning, colour, font, etc. All of this information is represented as a `windowSpec` description which is turned into an actual running UI instance by Visualworks Smalltalk's UI Framework (Figure 5.1 number 5). This runtime UI is an instance of the `UIBuilder` class which groups all UI component instances and maintains a link to the underlying model (see Section 5.5). These UI component instances know about their visualisation properties such as colour, layout, etc. By sending Smalltalk messages to a UI component, its properties can be changed.

In addition to building a UI from its `windowSpec`, one can create a runtime UI by directly communicating with a `UIBuilder` instance. Whereas the `windowSpec` is used to create a static version of the UI, changing the UI at runtime is done through this direct communication. In addition this mechanism can be used to create a UI without the use of a `windowSpec`. For each UI component a `componentSpec` is created and added to the instance with a `add:` message. At that point, the `UIBuilder` creates the actual component based on this spec, and adds it to its set of components. For example, a button can be created by executing:

```
ActionButtonSpec new name: #test ;
                     setLabel: #ExampleButton ;
                     layout: (120@160 corner: 210@190)
```

This code creates a `spec object` of the type action button. It is given a name, label and layout. This spec object is turned into an actual UI component upon adding it to a `UIBuilder`.

5.4.2 From DEUCE to a UIBuilder Instance

As described in Chapter 4 developers specify a user interface with high-level declarative statements in DEUCE. They will use the Visualworks Smalltalk UI Painter tool to put components on a canvas (randomly) and assign them a unique name. These names are used in the declarative statements to specify visualisation logic, such as properties of components and layout relations between components, and behavioural logic, such as events and actions on components. For example, the following rules are part of the visualisation logic for the calculator example in Chapter 4.

```
containsComponent(standardCalculator, firstDigits).
containsComponent(standardCalculator, secondDigits).
containsComponent(standardCalculator, operators).
containsComponent(standardCalculator, display).

window(standardCalculator).

text(standardCalculator,Calculator).
text(one, 1).

minimumHeight(?comp, 40).

backgroundColour(one,veryLightGray).
```

DEUCE transforms the declarative specification into a running UI by constructing a runtime user interface object. This user interface will contain several windows, each of which is a `UIBuilder` instance.

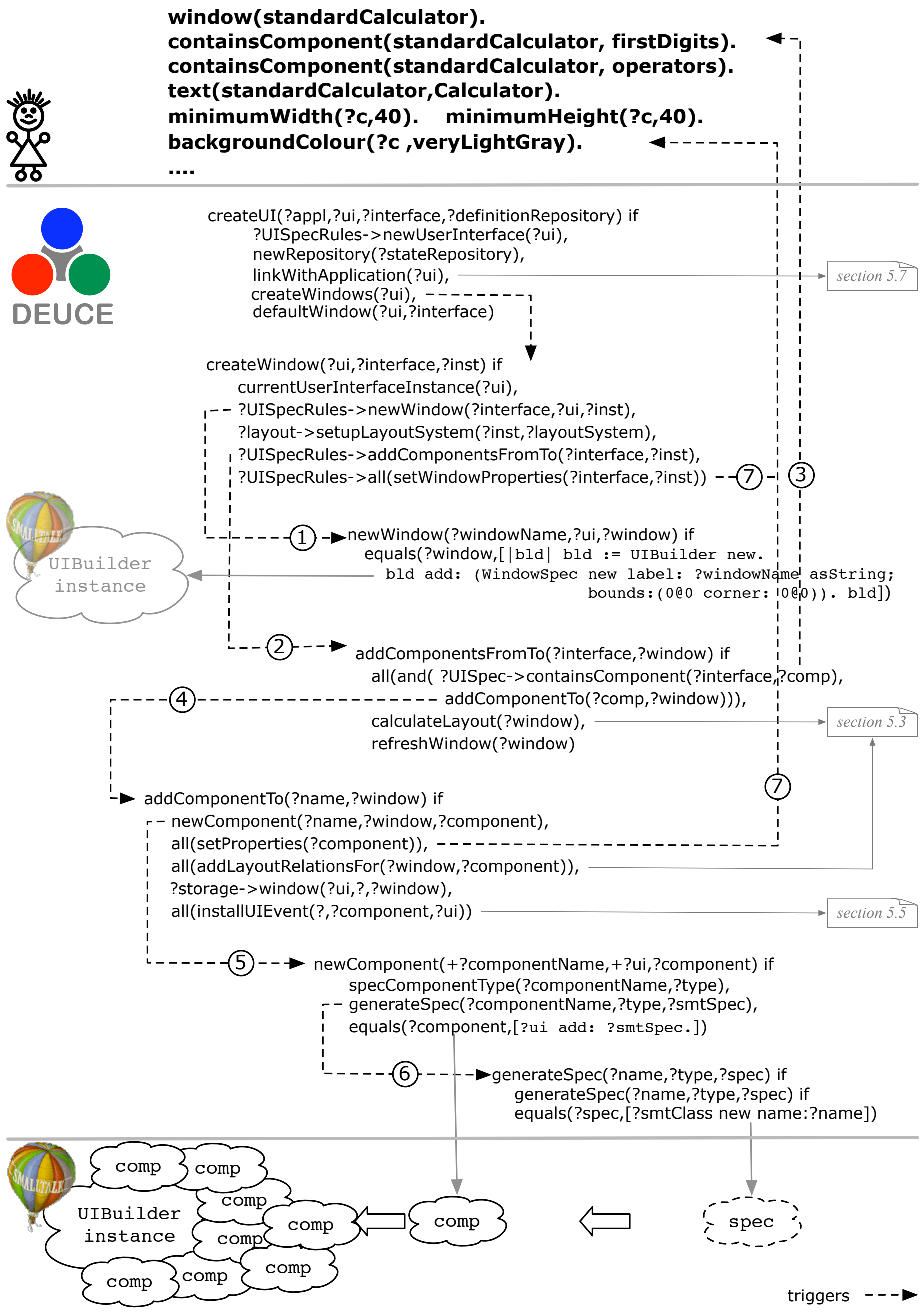
The process for creating windows is illustrated in Figure 5.5. The developers specify the high-level UI specifications which are used by DEUCE's internal rules (Figure 5.1 number 6) to create the actual UI. Note that by applying other rules later on, this window instance can be changed. The use of these rules by the declarative reasoning mechanism ensures the dynamic character of the UI.

First of all, a new `UIBuilder` instance is created (Figure 5.5 number 1) by the following rule.

```
newWindow(?windowName,?ui,?window) if
   equals(?ui,[|builder| builder := UIBuilder new.
               builder add: (WindowSpec new label: ?windowName asString;
                                          bounds: (0@0 corner: 0@0)).
               builder])
```

The only component that is part of this initial `UIBuilder` instance is a window component (i.e. a component of type `WindowSpec`). The properties of this window, such as colour and size, will be changed later on by sending Smalltalk messages. This empty window is needed in order to be able to add components to the UI. Therefore we chose to add this window by default upon creating the `UIBuilder` instance. The window properties are set with (Figure 5.5 number 7):

```
setWindowProperties(?windowName,?window) if
   setWindowSize(?windowName,?window).
```



```
window(standardCalculator).
containsComponent(standardCalculator, firstDigits).
containsComponent(standardCalculator, operators).
text(standardCalculator,Calculator).
minimumWidth(?c,40).    minimumHeight(?c,40).
backgroundColour(?c ,veryLightGray).
....
```

```
createUI(?appl,?ui,?interface,?definitionRepository) if
        ?UISpecRules->newUserInterface(?ui),
        newRepository(?stateRepository),
        linkWithApplication(?ui),
        createWindows(?ui),
        defaultWindow(?ui,?interface)

createWindow(?ui,?interface,?inst) if
     currentUserInterfaceInstance(?ui),
     ?UISpecRules->newWindow(?interface,?ui,?inst),
     ?layout->setupLayoutSystem(?inst,?layoutSystem),
     ?UISpecRules->addComponentsFromTo(?interface,?inst),
     ?UISpecRules->all(setWindowProperties(?interface,?inst))

newWindow(?windowName,?ui,?window) if
   equals(?window,[|bld| bld := UIBuilder new.
     bld add: (WindowSpec new label: ?windowName asString;
                               bounds:(0@0 corner: 0@0)). bld])

addComponentsFromTo(?interface,?window) if
   all(and( ?UISpec->containsComponent(?interface,?comp),
            addComponentTo(?comp,?window))),
   calculateLayout(?window),
   refreshWindow(?window)

addComponentTo(?name,?window) if
   newComponent(?name,?window,?component),
   all(setProperties(?component)),
   all(addLayoutRelationsFor(?window,?component)),
   ?storage->window(?ui,?,?window),
   all(installUIEvent(?,?component,?ui))

newComponent(+?componentName,+?ui,?component) if
   specComponentType(?componentName,?type),
   generateSpec(?componentName,?type,?smtSpec),
   equals(?component,[?ui add: ?smtSpec.])

generateSpec(?name,?type,?spec) if
   generateSpec(?name,?type,?spec) if
   equals(?spec,[?smtClass new name:?name])
```

Figure 5.5: From high-level user interface specification to Smalltalk user interface

```
setWindowProperties(?windowName,?window) if
   setWindowName(?windowName,?window).

setWindowSize(?name,?window) if
   ?UISpec->windowWidth(?name,?w),
   ?UISpec->windowHeight(?name,?h),!,
   setWindowSize(?window,?w,?h).

setWindowName(?windowName,?window) if
   ?UISpec->text(?windowName,?name),
   windowName(?window,?name)
```

For each of the properties the corresponding information is retrieved from the high-level UI specification. Through the use of a Smalltalk message the property of the actual window is set. An alternative rule to set the window's size to be as large as necessary to show all components, is the following:

```
setWindowSize(?name,?window) if
   expandedWindowSize(?window)

expandedWindowSize(?ui) if
   [xcoord := 0. ycoord := 0.
   ?ui namedComponents associationsDo:
      [:el | |cr| cr := el value component layout corner.
      (cr x > xcoord) ifTrue: [xcoord := cr x].
      (cr y > ycoord) ifTrue: [ycoord := cr y]].
   ?ui window minimumSize:  xcoord @ ycoord. true]
```

This rule will be triggered if no size or height was specified. The corner of the window is calculated based on the x and the y coordinate of the left-most and bottom-most component(s). Note that if the window does not contain any components, its size will be zero.

After creating the initial UIBuilder object, all components used in the interface (Figure 5.5 number 2) are added to it. To do so, for each component that is to be part of the UI (Figure 5.5 number 4), a corresponding Smalltalk object is created and added to the UI instance (Figure 5.5 number 5 and 6). How components are added to the UI, is shown in the next code snippet.

```
addComponentsFromTo(?interface,?window) if
   all(and(isComponentInInterface(?comp,?interface),
           addComponentTo(?comp,?window))),
   calculateLayout(?window),
   refreshWindow(?window).

addComponentTo(?name,?window) if
```

```
7      newComponent(?name,?window,?component),
8      all(setProperties(?component)),
9      all(addLayoutRelationsFor(?window,?component)),
10     ?storage->window(?ui,?,?window),
11     all(installUIEvent(?,?component,?ui))
```

Remember that the developers specified at the declarative level what components are part of an interface (line 2). Each component in this interface is added to the UI (line 3), the UI's layout is recalculated for this new interface (line 4), and its window is updated on the screen (line 5). Details on the layout mechanism is given in Section 5.3.

For each component that needs to be added, an actual Smalltalk component is generated and added to the UI by calling the `newComponent` rule (line 7). Its initial properties are set (line 8), the layout relations for the new components are added to the layout system (line 9) and the necessary events are installed (line 11) on the corresponding UI which the window belongs to (line 10). Details on installing the UI events can be found in Section 5.5. DEUCE now continues to operate on the runtime UI, namely this `UIBuilder` instance object.

Creating a Smalltalk UI component is done by the following rule:

```
1   generateSpec(?name,?type,?spec) if
2      smalltalkClassForSpecType(?type,?smtClass),
3      equals(?spec,[?smtClass new name:?name]).

4   newComponent(+?componentName,+?ui,?component) if
5      specComponentType(?componentName,?type),
6      generateSpec(?componentName,?type,?smtSpec),
7      equals(?component,[?ui add: ?smtSpec.])
```

When adding a new component to a `UIBuilder` instance, this component should be represented by a `spec` object. This spec object is a Smalltalk entity representing a component and its properties. Note that at this point it is not an actual UI component yet but it will turned into one by the `UIBuilder` upon addition. The rule on lines 1–3 creates a spec object for a component with a certain `?name` and `?type`. The second rule generates the actual component by calling this first rule (line 6). The spec is added to the runtime UI (line 7) and by doing so it is transformed into an actual UI component.

The above two rules generate Smalltalk components and should therefore now what Smalltalk class should be instantiated when creating such a component. The rule `smalltalkClassForSpecType` retrieves the corresponding Smalltalk class (line 2). The rule `specComponentType` will query the UI specification to know what kind of component is needed (line 5). Below both rules are shown for labels, inputfields and buttons.

```
1   smalltalkClassForSpecType(label,[LabelSpec]).
2   smalltalkClassForSpecType(inputField,[InputFieldSpec]).
3   smalltalkClassForSpecType(ActionButtonSpec,[ActionButtonSpec]).
```

```
specComponentType(?compName,label) if
   ?UISpec->label(?compName)
specComponentType(?compName,inputField) if
   ?UISpec->inputField(?compName)
specComponentType(?compName,button) if
   ?UISpec->button(?compName)
```

DEUCE provides rules to set the different properties for the Smalltalk UI components. For instance, the following rule sets the text properties. The rule `setText` (line 1) retrieves the text property from the UI specification by triggering the `specText` rule (line 3) and sets the property with the `text` rule (line 4). Note that a Smalltalk inputField component (line 13) needs to be sent a different message than a Smalltalk label component (line 9) in order for its text to be changed. Because of these abstraction rules developers no longer need to be aware of this low-level difference if DEUCE is used.

```
1   setText(?component) if
2      name(?component,?name),
3      specText(?name,?text),
4      text(?component,?text).

5   specText(?compName,?text) if
6      ?UISpec->text(?compName,?text).

7   text(?component,?text) if
8      smalltalkComponentType(?component,label),!,
9      [?component widget labelString: (?text asString).
10     ?component widget invalidate. true].

11  text(?component,?text) if
12     smalltalkComponentType(?component,inputField),!,
13     [?component widget editText: (?text asString).
14     ?component widget model value: ?text .
15     ?component widget invalidate. true]
```

5.5 User Interface Events

When a user performs an action on the user interface, for instance by clicking a button or by typing text, the user interface launches an event to notify the underlying system that an event has occurred. The system is responsible for taking the appropriate actions, either by calling the user interface or by calling the application code. Visualworks Smalltalk provides an event handler mechanism to deal with such events (Figure 5.1 number 5c). When creating a high-level UI specification, developers specify the events that can occur in a user interface and the logic actions that are to be triggered upon such an event. DEUCE transforms this high-level specification into the event handling mechanism by providing a set of rules as part of its core logic (Figure 5.1 number 6).

5.5.1 The Visualworks Smalltalk `applicationModel`

Chapter 2 illustrated how Visualworks Smalltalk uses the MVC pattern to specify user interfaces and to separate the application (model) from its graphical user interfaces (views). Visualworks Smalltalk uses an ‘application model’ as an intermediate to handle the communication between model and view. It contains the state and behaviour of the UI, as well as its link to the application. Application data can be monitored in the application model through the use of value holders. When changing the monitored value (i.e. application data) through the value holder, the value holder assures that all its dependents are notified of the change such that these can undertake the necessary actions. At this point the application data is not aware of who depends on it and to whom a notify message is sent. Note however that application data is changed through the value holder when a change originates from within the application model (and hence the UI). If changes happen from outside the UI, the value holder is not used, and therefore its dependents are not notified. In order to notify the UI (application model) anyway, the application methods need to include the notify statement.

When creating a user interface in Visualworks Smalltalk, the developers are responsible for implementing a subclass of the `applicationModel` class which specifies the UI actions and relations with the application. Relations between UI and application are specified through the value holder mechanism discussed above, and by accessing the application directly through message sending. UI actions are methods that will be triggered upon UI events, such as getting and losing focus, notification, validation, and closing a window.

When specifying a UI component with the UI Painter Tool (see Section 5.4), the developers provide the name of such an action method to function as the ‘model’ for that component. Such a model is triggered upon one event only. For instance for a button, the model only gets triggered when that button is clicked. Nonetheless, additional events can be added programmatically by using a `when:send:to:` registration message (Figure 5.1 number 5c). This message is sent to a UI component with the following arguments:

- the name of an event. The events that can be understood by a UI component are predefined by Visualworks Smalltalk.
- the name of the method to be sent upon this event.
- the object which the message should be sent to. Typically this is the ‘self’ argument as these registration methods are set up in the application model and the application model contains the methods specifying the UI behaviour.

Note that for each event, whether it is registered as the model of a component or through an event registration method, there is a corresponding method in the application model that contains the code to be executed upon that event.

Upon triggering an event, the event calls the method in the `applicationModel` whose name was passed on upon installing the event handler. The `applicationModel` can be omitted by replacing such a method name with a Smalltalk block (i.e. executable pieces of Smalltalk code). This block contains the code to execute upon the event, and thus to handle the event. This allows for setting up a user interface in Visualworks Smalltalk without using an `applicationModel`.

For instance, setting the model of a button to a Smalltalk block, will result in the code in the block to be executed upon clicking the button:

```
aButton model: [Transcript show: ‘Button #test has been clicked’]
```

A Smalltalk block is an object and can be executed by sending it the `value` message. Therefore, it can be used in event registration methods as follows:

```
aButton when: #clicked
        send: #value
        to: [Transcript show: ‘Button #test has been clicked’]
```

Note that this code achieves the same as the first example, but as the `model` of a component is linked to one type of event only, it is rather limited and other events need to be set up with code like the second example.

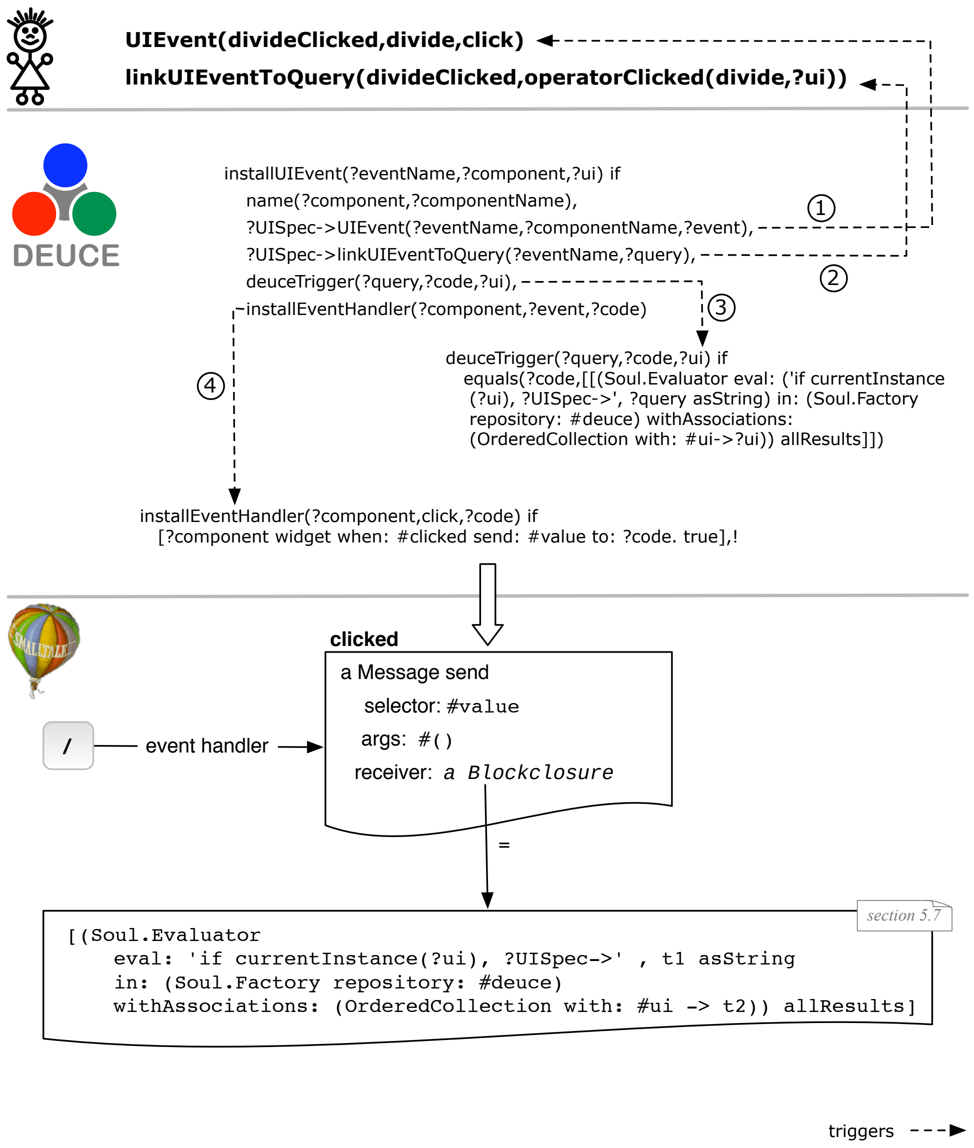
5.5.2 Linking UI Events With a Query

In Chapter 4 we have explained that part of the high-level presentation logic states what events can be sent to a UI component and what behaviour, or query, is linked with this event. For example, the divide button can receive a `click` event, upon which the `operatorClicked` query is launched. This is specified with:

```
UIEvent(divideClicked,divide,click).

linkUIEventToQuery(divideClicked,operatorClicked(divide,?ui))
```

DEUCE provides a set of rules at the logic level (Figure 5.1 number 6) to transform these statements into the Visualworks Smalltalk event mechanism (Figure 5.1 number 5c) which was explained above. This internal process is illustrated in Figure 5.6. Based on the high-level specification for the event (Figure 5.6 number 1) and corresponding query (Figure 5.6 number 2), an event handler is created for the component (Figure 5.6 number 4). This handler will execute the code that launches the particular query in DEUCE (Figure 5.6 number 3).

The following rule installs the necessary code fragment (Smalltalk block) as an event handler on the corresponding UI component. When an event is sent to that component, it will execute the code fragment. This code will be responsible for calling DEUCE



```
UIEvent(divideClicked,divide,click)
linkUIEventToQuery(divideClicked,operatorClicked(divide,?ui))
```

```
installUIEvent(?eventName,?component,?ui) if
   name(?component,?componentName),
   ?UISpec->UIEvent(?eventName,?componentName,?event),
   ?UISpec->linkUIEventToQuery(?eventName,?query),
   deuceTrigger(?query,?code,?ui),
   installEventHandler(?component,?event,?code)
```

```
deuceTrigger(?query,?code,?ui) if
   equals(?code,[[(Soul.Evaluator eval: ('if currentInstance
    (?ui), ?UISpec->', ?query asString) in: (Soul.Factory
    repository: #deuce) withAssociations:
    (OrderedCollection with: #ui->?ui)) allResults]])
```

```
installEventHandler(?component,click,?code) if
  [?component widget when: #clicked send: #value to: ?code. true],!
```

```
a Message send
   selector: #value
   args: #()
   receiver: a Blockclosure
```

```
[(Soul.Evaluator
    eval: 'if currentInstance(?ui), ?UISpec->' , t1 asString
    in: (Soul.Factory repository: #deuce)
    withAssociations: (OrderedCollection with: #ui -> t2)) allResults]
```

Figure 5.6: From high-level user interface event specification to Smalltalk event handler

and triggering the query that was specified at the high-level. In this rule, the code fragment is generated with the rule `deuceTrigger` (line 5). Details of this mechanism are discussed in Section 5.8.

```
1  installUIEvent(?eventName,?component,?ui) if
2     name(?component,?componentName),
3     ?UISpec->UIEvent(?eventName,?componentName,?event),
4     ?UISpec->linkUIEventToQuery(?eventName,?query),
5     deuceTrigger(?query,?code,?ui),
6     installEventHandler(?component,?event,?code).

7  installEventHandler(?component,click,?code) if
8     [?component widget when: #clicked send: #value to: ?code. true]

9  installEventHandler(?component,getFocus,?code) if
10    [?component widget when: #gettingFocus send: #value to: ?code. true].
```

The high-level specification is retrieved on lines 3–4. The `deuceTrigger` rule is called on line 5 to generate the Smalltalk code fragment that will call DEUCE upon execution. The `installUIEvent` rule installs the fragment as an event handler on the UI component (line 8). Note that different events require different handlers. For instance, a click event handler (line 7) is different from an event handler for dealing with a component getting the focus (line 9) since the first sends the `clicked` event and the second the `gettingFocus` event. Again by using DEUCE we are able to abstract away from these low-level Smalltalk details. The developers will no longer be confronted with this and can simply specify this behaviour in terms of the `getFocus` and `click` facts.

5.6 User Interface and Application Actions

Both UI events (e.g. mouse click) and application events (e.g. data change), will trigger actions. UI actions will for instance change the user interface by updating layout, adding or removing components and changing component properties. Application actions trigger behaviour of the underlying application.

In DEUCE the application is implemented in Visualworks Smalltalk by the developers and instantiated by DEUCE. The running UI created by DEUCE is a `UIBuilder` instance. Hence, both the application instance and the UI instance are Smalltalk objects. Performing actions on both boils down to sending a Smalltalk message to the corresponding object (Figure 5.1 number 8 and 9).

For example, disabling the equals button is expressed with the statement on line 1 in the code snippet below. Lines 2–4 translate this in sending the `disable` message to the component with name `equals` in the current user interface bound to the `?ui` logic variable.

```
disable(equals,?ui).

disable(?compName,?ui) if
        componentWithNameIn(?component,?compName,?ui),
        [?component disable. true]
```

The same applies for an application action, for instance when computing the result in the calculator example.

```
computeResult(+?appl) if
   [?appl compute. true]
```

5.7 Application Events

In order for the UI and application to interact with each other, the application needs to be able to trigger behaviour in the UI. The places at which the application needs to call the UI, are what we call application events. These events are specified in the application logic concern. Nevertheless the application itself will need to be instrumented with appropriate code such that the UI actually gets called. Therefore DEUCE will translate the application events into instrumentation code (Figure 5.1 number 11). As a mechanism to add this piece of code to the application automatically, DEUCE uses method wrappers (Figure 5.1 number 10). In what follows we discuss the choice of method wrappers as well as the actual transformation from high-level specification to method wrapper.

5.7.1 Method Wrappers

Method Wrappers have been introduced by Brant et al. [Bra98] to add hidden behaviour to a method without recompiling it. Typically they are used to change a method's behaviour. They allow adding behaviour before, after or around the default method execution [Bra98]. As DEUCE installs the necessary method wrapping mechanism behind the scenes, the developers are screened from the details of using method wrappers.

When using the method wrapper framework as provided by Brant, developers create their own dedicated subclass of the class `MethodWrapper` and redefines the `beforeMethod` and/or `afterMethod` to add specific behaviour for the wrapper. To use the method wrapper, the class is sent the `on:inClass:` message. The first argument is the selector for the method, and the second is the class that defines the method. This returns a new method wrapper which can then be installed the by sending the `install` message and uninstalled by sending `uninstall` message. If the method wrapper is installed, its before and/or after method are executed when the method (first argument) is sent to the class (second argument).

5.7.2 Linking Application Events With a Query

Recall from Chapter 4 that developers specify, as a part of the application logic concern, what the application events are and how these are linked to a logic query. For example,

```
1  applicationEvent(resultChanged,calculator,#result:).
2  role(calculator,[Deuce.Calculator]).

3  linkApplicationEventToQuery(resultChanged,updateResult(?ui))
```

The internal process to transform this high-level specification into the actual method wrapper behind the corresponding application method, is illustrated in Figure 5.7. Based on the information specified in the high-level specification, the piece of code that will launch the specified query (Figure 5.7 number 1) in DEUCE is generated (Figure 5.7 number 5) and installed as a method wrapper (Figure 5.7 number 4) on the corresponding application class (Figure 5.7 number 3) for its specified selector (Figure 5.7 number 2).

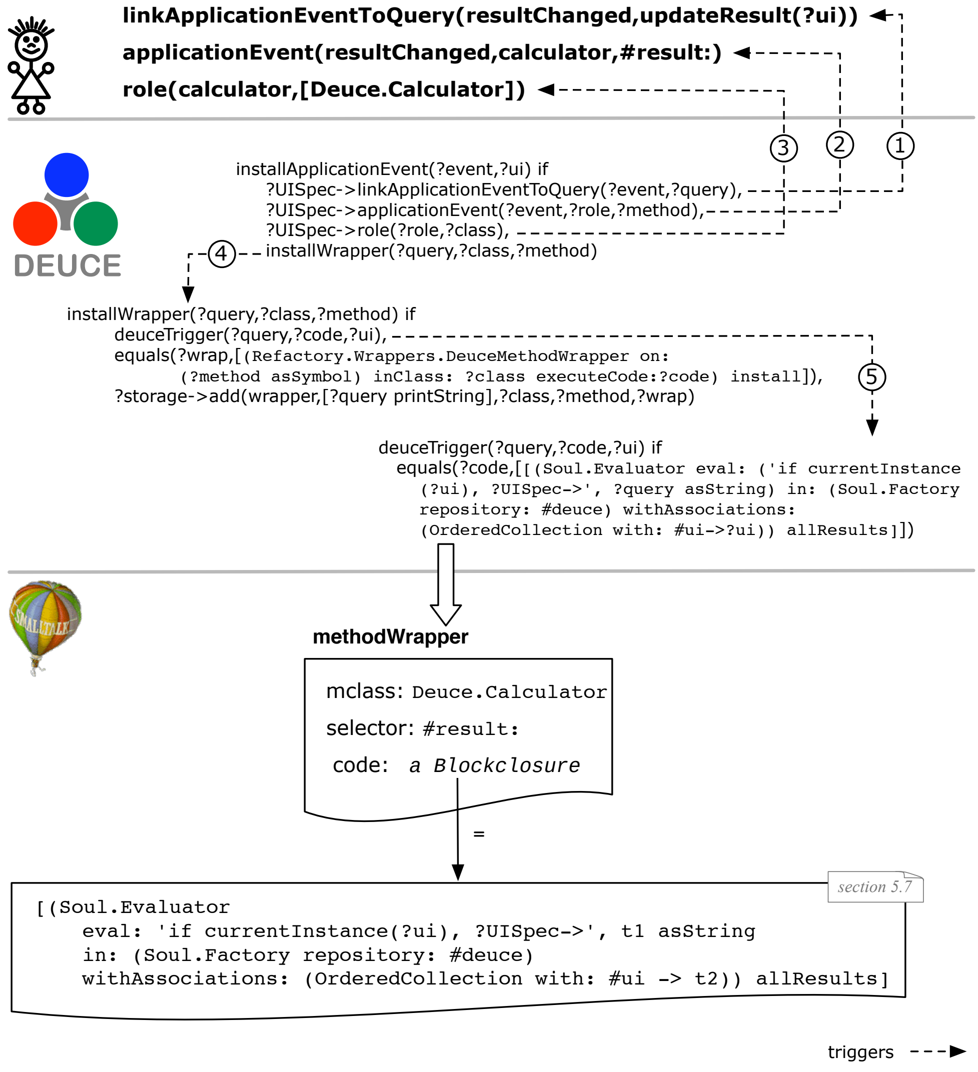
For each of the `linkApplicationEventToQuery` specifications, DEUCE installs a method wrapper to trigger the corresponding query after the application event occurred. This is done with the following rule:

```
1  installApplicationEvent(?event,?ui) if
2     ?UISpec->linkApplicationEventToQuery(?event,?query),
3     ?UISpec->applicationEvent(?event,?role,?method),
4     ?UISpec->role(?role,?class),
5     installWrapper(?query,?class,?method).
```

The `applicationEvent` (line 3) specifies what method in the application corresponds to the event. The `role` (line 4) specifies the class in the application to which this method belongs. The `installWrapper` (line 5) rule installs the actual method wrapper as follows:

```
1  installWrapper(?query,?class,?method) if
2     get(wrapper,[?query printString],?class,?method,?wrapper),!.

3  installWrapper(?query,?class,?method) if
4          deuceTrigger(?query,?code,?ui),
5          equals(?wrapper,[ (Refactory.Wrappers.DeuceMethodWrapper
6         on: (?method asSymbol)
7         inClass: ?class
8                  executeCode:?code) install.]),
9          add(wrapper,[?query printString],?class,?method,?wrapper)
```

Installed method wrappers are kept track of (line 2) such that methods are only wrapped once for a certain application event. This is important because otherwise a method would be wrapped for all UI instances, and upon that particular application event



```
linkApplicationEventToQuery(resultChanged,updateResult(?ui))
applicationEvent(resultChanged,calculator,#result:)
role(calculator,[Deuce.Calculator])
```

```
installApplicationEvent(?event,?ui) if
    ?UISpec->linkApplicationEventToQuery(?event,?query),
    ?UISpec->applicationEvent(?event,?role,?method),
    ?UISpec->role(?role,?class),
    installWrapper(?query,?class,?method)
```

```
installWrapper(?query,?class,?method) if
    deuceTrigger(?query,?code,?ui),
    equals(?wrap,[(Refactory.Wrappers.DeuceMethodWrapper on:
          (?method asSymbol) inClass: ?class executeCode:?code) install]),
    ?storage->add(wrapper,[?query printString],?class,?method,?wrap)
```

```
deuceTrigger(?query,?code,?ui) if
  equals(?code,[[(Soul.Evaluator eval: ('if currentInstance
    (?ui), ?UISpec->', ?query asString) in: (Soul.Factory
    repository: #deuce) withAssociations:
    (OrderedCollection with: #ui->?ui)) allResults]])
```

```
mclass: Deuce.Calculator
selector: #result:
code:  a Blockclosure
```

```
[(Soul.Evaluator
    eval: 'if currentInstance(?ui), ?UISpec->', t1 asString
    in: (Soul.Factory repository: #deuce)
    withAssociations: (OrderedCollection with: #ui -> t2)) allResults]
```

Figure 5.7: From high-level application event specification to low-level method wrapper

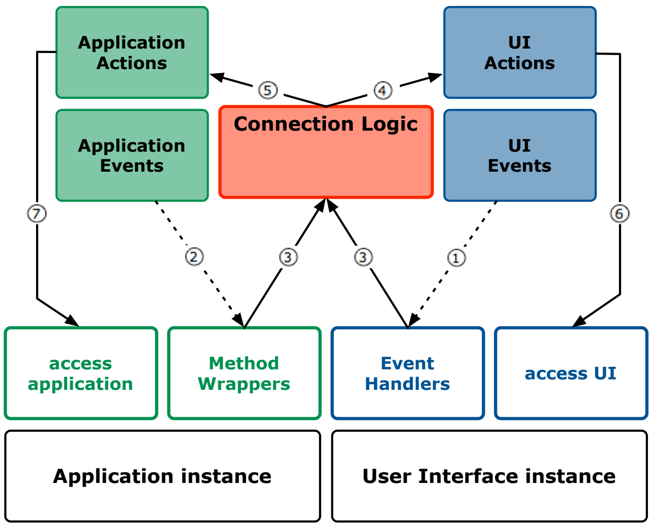


Figure 5.8: Linking application and user interface

DEUCE would be triggered more than once. If the method has not been wrapped yet for the application event, the second rule applies (lines 3–9). Details on how the piece of code to trigger DEUCE is generated (line 4), is explained in Section 5.8. This code is installed as a method wrapper (line 5) and stored for further reference (line 9).

5.8 Connecting User Interface and Application

In order to bring presentation logic and application logic together, the developers provide the connection logic (Figure 5.1 number 14). This connection logic specifies logic queries that connect UI and application. How this mechanism works and how it is installed by DEUCE, is explained in this section.

5.8.1 Connection Logic Queries

Both the presentation logic as the application logic specify events and actions. Events are occurrences in the UI or in the application that will trigger some changes in either the UI or the application. These changes are called actions. Hence, events trigger actions. With DEUCE, the connection logic acts as an intermediate between events and actions, such that for example UI events do not trigger application actions directly. If this would be otherwise, UI events would know about application actions, which violates the principle of separation of concerns. The connection logic adds an extra level of abstraction such that either application or presentation logic can be changed without affecting the other. This mechanism is illustrated by Figure 5.8.

The UI specification of UI events is transformed into event handlers (Figure 5.8

number 1) as was explained in Section 5.5. The application events are transformed into method wrappers (Figure 5.8 number 2) as was shown in Section 5.7. Triggering an event results in calling a logic query (Figure 5.8 number 3), which is part of the connection logic. These queries can trigger either or both UI actions (Figure 5.8 number 4) and application actions (Figure 5.8 number 5). UI actions result in accessing the UI by sending a Smalltalk message to the UI instance (Figure 5.8 number 6). Application actions result in accessing the application by sending a Smalltalk message to the application instance (Figure 5.8 number 7).

Connection queries are specified by UI developers. For example the `operatorClicked` rule from Chapter 4:

```
operatorClicked(equals,?ui) if
   application(?ui,?appl),
   computeResult(?appl,?result),
   buttonStates(equals,?ui)
```

This rule specifies that when solving the query `operatorClicked`, which will be launched upon clicking an operator button in the calculator example, the `computeResult` and `buttonStates` rules are fired. The `computeResult` rule is specified as part of the application logic whereas the `buttonStates` rule is specified as part of the presentation logic.

5.8.2 Launching a Query in DEUCE

As connection queries are launched upon UI and application events, the (Smalltalk) mechanisms to deal with these events launch a query in DEUCE. Launching a query in DEUCE can be done from within Smalltalk by calling the SOUL evaluator. For instance the following launches the query to create a new UI from within one of the DEUCE tools:

```
 1  |query assoc appl rep|

 2  appl := self currentApplication.
 3  rep  := self currentRepository.
 4  assoc := OrderedCollection with:#application->appl
 5                             with:#specRepository->rep.

 6  query := 'if createUI(?ui, ?application, ?specRepository)'.

 7  (Soul.Evaluator
 8     eval: query
 9     in: (Soul.Factory repository: #deuce)
10     withAssociations: assoc) nextResult.
```

Calling the SOUL evaluator is expressed on lines 7–10. The call has as arguments a query (in string format) to evaluate (line 8) and a repository to evaluate the query in (line 9). In this case the query 'if createUI(?ui, ?application, ?specRepository)' is evaluated in the deuce repository. Additionally one can bind, or associate, variables in the query to Smalltalk objects (line 10). Here the ?application and ?specRepository variables are bound to Smalltalk objects (being the results from the methods currentApplication and currentRepository). The variable ?ui is unbound at the time of evaluating the query in the SOUL Evaluator.

At the DEUCE level, the rule deuceTrigger shown below uses this mechanism to generate the piece of code ?code to make such a call to the SOUL evaluator. Note that SOUL requires all variables used in a Smalltalk statement to be bound before evaluating this block. Hence, the variables ?query and ?ui should be bound when triggering this rule, since they are used within the Smalltalk statement.

```
deuceTrigger(?query,?ui,?code) if
   equals(?code,[[(Soul.Evaluator
         eval: ('if ?UISpec->', ?query asString )
         in: (Soul.Factory repository: #deuce)
         withAssociations:(OrderedCollection with: #ui->?ui))
      nextResult]])
```

This rule deuceTrigger is used by DEUCE when installing UI events and application events, such that both call upon DEUCE when triggered.

5.9 Discussion of the Mechanisms Behind DEUCE

When implementing DEUCE we opted for certain implementation techniques as these techniques provide the necessary infrastructure to fulfill the postulated requirements for a solution for separation user interface concerns. In this section we recapitulate the requirements and how they are fulfilled by DEUCE. Next we discuss some of the strengths and limitations that resulted from choosing these techniques.

5.9.1 Fulfilment of the Requirements

In Chapter 3 the requirements for a solution for achieving a separation of user interface concerns were formulated. In DEUCE several mechanisms are used in order to fulfill these requirements. Using SOUL as a declarative medium allows for a separate specification for each concern (requirement 1) at both a high and low-level of abstraction (requirement 2) which are mapped onto each other (requirement 3) by the declarative reasoning mechanism. This reasoning mechanism also supports the automatic composition of the several concerns into the final application (requirement 4). Additionally, this last requirement is supported by SOUL's symbiosis with Smalltalk which allows for installing the necessary linking mechanisms in terms of event handlers and method wrappers. An automated layout (requirement 5) is supported by the Cassowary linear constraint solver.

5.9.2 The Power of SOUL's Symbiosis With Smalltalk

One of the strengths of SOUL is its symbiosis with Smalltalk which allows to switch back and forth between the Smalltalk and logic level. The latest symbiosis mechanisms added to SOUL introduce an even higher transparency between both levels by allowing to write logic statements in the form of keyword based messages which is the syntax that is also being used for Smalltalk messages. At the moment SOUL supports both this new syntax as the more 'classical' syntax for writing its logic statements. Note that throughout this dissertation and our implementation we use the 'classical' syntax although in future work one might opt for more transparency between logic and Smalltalk.

As a consequence of SOUL's symbiosis with Smalltalk, DEUCE has access to the underlying Smalltalk applications and user interfaces. In combination with Smalltalk's dynamic environment, this allows DEUCE to interact with the running application and adapt UIs dynamically where necessary. Using SOUL has provided us for rapid prototyping DEUCE, which from a researchers point of view allowed us to focus on the separation of user interface concerns.

5.9.3 Modularising Logic

SOUL uses logic layers and repositories (see Chapter 4) to modularise logic. However, improvement to the implementation of these structures would enrich this modularisation.

Logic layers for Separating Concern Specifications In SOUL, logic facts and rules are stored in layers (i.e. static logic databases). These layers allow to modularise the UI concerns in two dimensions. Firstly, UI concerns are modularised by dedicating a separate layer of logic facts and rules to each of the concerns. Hence changing or evolving a concern is limited to changing its particular layer(s). Secondly, within each concern several layers can be used to represent reusable sets of logic. For instance, the standard and the scientific calculator have the same visualisation for the number buttons. By putting the specification of their properties and layout into a separate logic layer (i.e. rule-base), it can be reused for both calculators. Additionally, the layer modularisation is used by DEUCE to provide different levels of abstraction. The UI developers specify the high-level UI in layers, DEUCE provides layers for the mapping from high-level to low-level, for the layout solving mechanism, for the mapping from (mid-level) components to actual Smalltalk components, etc.

This layer modularisation is a particularity of SOUL and so far it was sufficient for our purposes. However, when introducing large sets of rules and facts, a kind of rule-management system will be required in order to select the relevant rule layers and plug in and out the applicable logic. As a future work we consider to look into the rule management used in other large scale declarative systems.

Repositories for selecting Relevant Rule-bases The facts and rules from the layers are loaded into a repository (i.e. a runtime logic database) when used for reasoning. One can select what layers to load into a repository. By doing so, only the layers that are relevant for a certain UI specification are loaded and used during the reasoning process. The repository represents the scope within which the predicates of that repository can be evaluated. Through repository variables (denoted by `?repositoryVariable->`) one can specify what repository, and thus what scope, to use when evaluating a predicate. Additionally, when new facts or rules are asserted at runtime, this information is added to the repository as this is the runtime knowledge base.

The downside of using repositories to assert new facts, is a possible loss of information. When one of the loaded layers changes, for instance by extending an existing UI specification with a new layout strategy, the repository is re-instantiated in order to incorporate this new behaviour. However, this also results in the loss of all information that was asserted to the repository at runtime and that is not part of a layer. Nevertheless it is desirable for some runtime information to survive a re-instantiation. For example the state of the runtime UI should be preserved as long as the UI is running, even when its specification changed during that time. Therefore this kind of information should not be asserted to the current repository but should be stored in either a separate repository or a separate Smalltalk state object.

5.9.4 A Separate Reasoner Behind Every UI

With the rise of context-sensitive systems, the research community has expressed the need for accessing UI specification knowledge at runtime [Dem06]. This includes access to the power of the declarative reasoning process to infer the necessary conclusions and consequences based on these UI specifications. Therefore each UI instance should have access to its specification and to the reasoning process. Ideally each UI instance has its own reasoner that remains accessible during the entire lifetime of that UI.

Currently DEUCE does not use multiple SOUL evaluators in coexistence. For every query that is to be evaluated, a new evaluator object is created. Additionally in SOUL the runtime knowledge that has to remain available in between several evaluations (i.e. reasonings) is stored in a factory object through the structure of repositories. Repositories provide a scope of evaluation by means of repository variables. These variables can be rebound to other repositories programmatically. DEUCE uses this mechanism to switch between UI specifications, but future work should consider re-implementing part of SOUL and DEUCE.

As a UI specification is represented by a repository, the core logic repository that acts as DEUCE's central engine, needs to plug in and out different specification repositories. This is achieved by rebinding the repository variable of the core logic repository that is referring to the specification repository. The reasoning process will then look up information (i.e. specifications) within the scope of this new repository. Recall from Chapter 3 that repository variables are defined within the originating repository and are bound to the other repositories. Rebinding these variables can be done both programmatically or through SOUL's repository browser.

5.9.5 The Effects of Replacing the Mechanisms Behind DEUCE

In figure 5.1 we have depicted the DEUCE framework, and what mechanisms DEUCE uses to transform the high-level UI specification into an actual running software system. DEUCE's core logic contains rules to map the high-level specifications onto the actual mechanisms being used. When replacing these mechanisms, part of these core logic rules will also need to be replaced. However, the rest of DEUCE's implementation and core logic will remain unaffected. For example, as shown in Figure 5.9 a, replacing the constraint solver mechanism which currently is used to achieve an automated layout, will require the developers to provide a new layout mechanism as well as a new set of logic rules that map the layout specification onto this new mechanism. The same is true when replacing the composition mechanism that links the application code with the UI (Figure 5.9 b) and when using another UI framework (Figure 5.9 c).

Currently DEUCE is implemented in Smalltalk and SOUL. As SOUL provides means to communicate with Java applications, it is possible to reuse large parts of DEUCE for Java applications. However, when using DEUCE for a different platform this requires the mechanisms behind DEUCE to be re-implemented. As a consequence, also its core logic rules will change as they will map the high-level UI specification onto different low-level mechanisms. The high-level UI specification however can be reused as it has no notion of the low-level mechanisms that are used. Remark however that changing the underlying application, implies its 'interface' (or hooks) to the UI changes accordingly. As a consequence the application events and application actions will also need to be re-implemented. The effects of changing to a different platform and a different application on DEUCE are shown in Figure 5.10.

5.10 Conclusion

In this chapter we have explained the mechanisms used by DEUCE to transform the high-level application into the final runtime system. This system consists of an application instance and an user interface instance. The application instance is an instantiation of the application developed in Visualworks Smalltalk. The user interface instance is a runtime Visualworks Smalltalk UI and consists of UIBuilder objects. Both instances are linked together through the logic level. This level is called by user interface and application events. The mechanism to trigger the logic level is installed through Smalltalk events and method wrappers and. Upon the event, a connection query is launched in DEUCE. This connection query can trigger both UI actions and application actions. Executing the actual action is achieved by sending a message to the Smalltalk objects that represent the application and the UI (and its components). Automatically laying out the UI components on the screen, is done by using a Cassowary linear constraint solver. We concluded this Chapter with some remarks on the chosen mechanisms and an explanation of how fulfill the requirements of the conceptual solution as was proposed

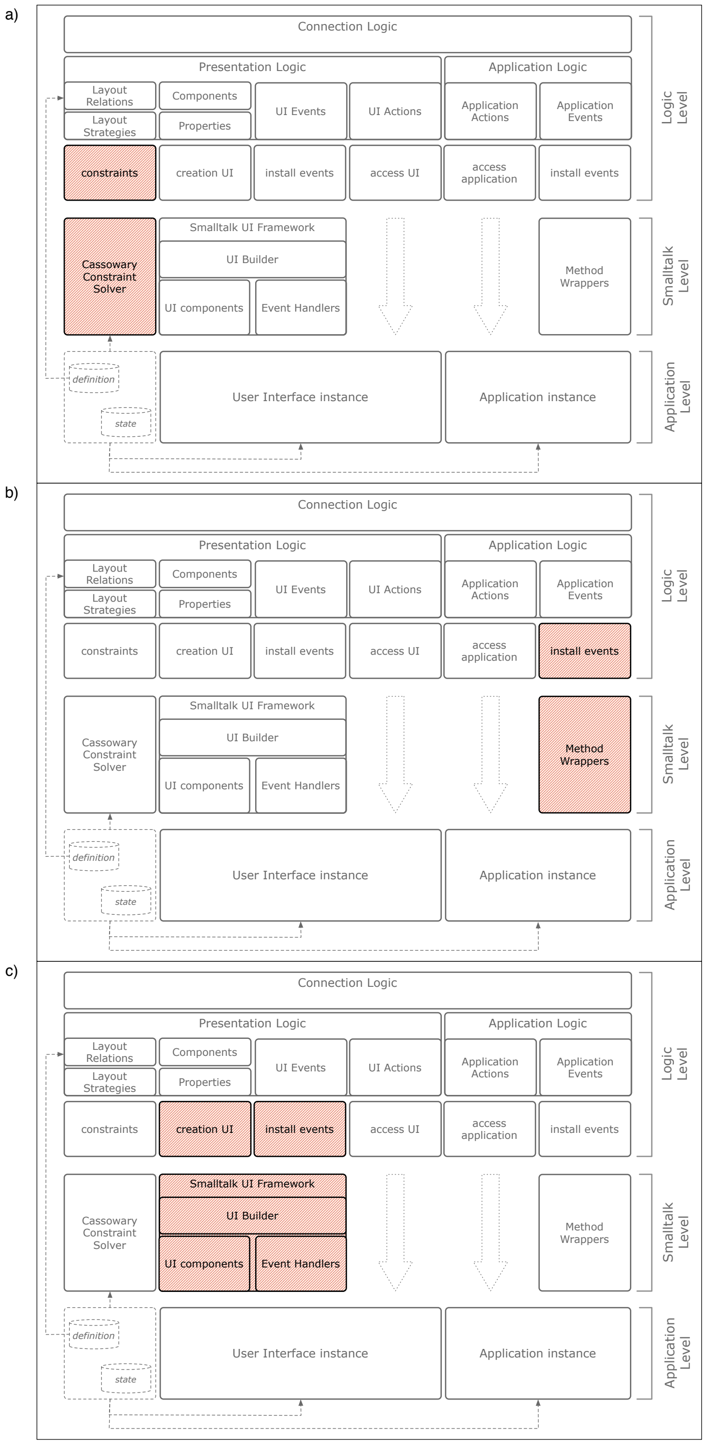


Figure 5.9: Replacing the mechanisms behind DEUCE: a) layout b) composition c) UI framework

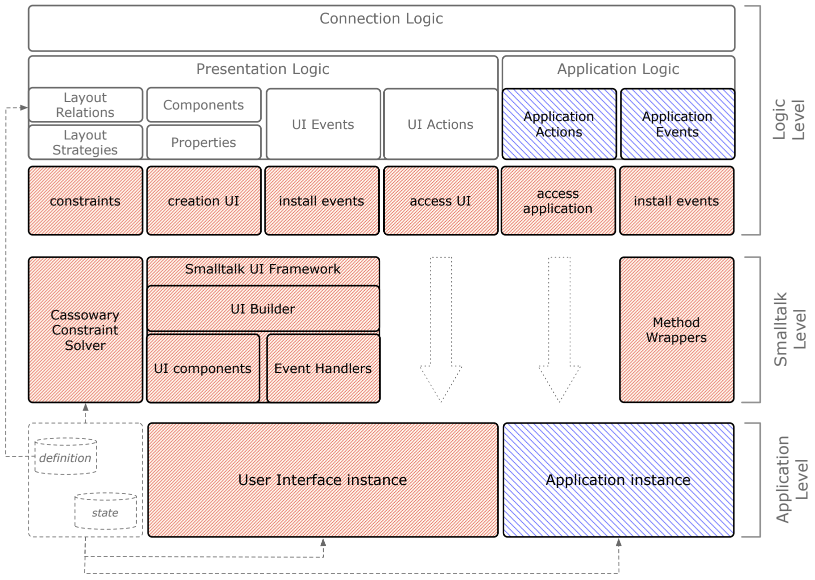


Figure 5.10: Using DEUCE on another platform

in Chapter 3. In Chapter 6 we will validate the conceptual solution of Chapter 3 and its practical implementation of Chapters 4 and 5 by illustrating that the separation of concerns is maintained when evolving each of the concerns.

Chapter 6

Validation With *MijnGeld*

In this dissertation we proposed a conceptual solution for achieving a separation of user interface concerns and we implemented a proof-of-concept DEUCE to support this solution. In what follows we will validate our approach by applying it to the *MijnGeld* case study. A first validation scenario in Section 6.2 will illustrate how UI specifications in DEUCE can be reused without requiring exhaustive changes as are required in the VisualWorks Smalltalk implementation. Section 6.3 shows how evolving the presentation logic concern does not affect the other concerns. Evolving the application logic concern in Section 6.4 also illustrates this separation as well as how changes are localised in that concern rather than being spread al over the application code. The scenarios in Section 6.5 and Section 6.6 focus on the connection logic concern. This concern is triggered by both UI events and application events respectively. Before going into details on these validations, Section 6.1 gives some general information about the *MijnGeld* application.

6.1 *MijnGeld*: a Personal Finance Application

As a case study for this validation we use the *MijnGeld*[1] application [Ven]. *MijnGeld* is a free Visualworks Smalltalk implementation for a personal finance manager, with support for online banking and native support for several banks. Its standard functionality includes adding accounts of different types, namely bank accounts, cash accounts, liability accounts, etc. For each of the accounts, one can add transaction statements, such as deposits, transfers and withdrawals. Although the information for both different types of bank accounts and different types of transactions are similar, each of the types has slight differences with the other types. Additionally one can keep track of payees and payment categories and perform searches based on the payee or payment category as well as based on date, amount, account, etc. Furthermore budgets can be created and evaluated. We explain some of this functionality in further detail as we go along.

[1] *MijnGeld* is Dutch for MyMoney.

Several beta versions of *MijnGeld* have been published, but the application is still a work in progress. It already offers a great deal of the finance management functionality but some of its parts are still left blank. Nevertheless *MijnGeld* counts at the moment 107 classes, 2289 methods and 36.618 lines of code. There is a clear distinction between its functionality and its user interfaces to interact with the application. It is implemented in a typical MVC style which uses an observer mechanism to inform the UI of changes in the application and an event-handling and data-binding mechanism to link the UI to the application. Out of the 36.618 lines of code, 16.501[2] lines are related to the UI code. Table 6.1 gives an overview of lines, classes and methods count for the *MijnGeld* system.

Table 6.1: Overview of the *MijnGeld* code base

	#lines	#classes	#methods
in total	36.618	107	2289
related to Application code	8852	47	723
related to UI code	16.501	53	1565
related to windowSpecs	14.236		81

As *MijnGeld* is implemented in VisualWorks Smalltalk, each of its sub-applications has its own `applicationModel` class. The `applicationModels` account for 36 of the classes of the MijnGeld system. These classes implement the initial UI visualisations (i.e. `windowSpecs`), their links to the application (i.e. model methods and event-handlers installations), the dynamic UI updates (e.g. changing labels at runtime), the control flow to decide what change to apply, etc. Basically the `applicationModel` provides for the different UI concerns. Unfortunately they are entangled. For example, almost 36% of the methods in `applicationModels` call upon both the `applicationModels` (UI) and the underlying domain classes of the application. 74 methods in the application classes (almost 12%) send a self changed message such that views (UIs) are notified. In table 6.2 we list additional examples of UI related code that indicates the code scattering and tangling that typically occurs in UI code.

Table 6.2: *MijnGeld* code statistics

		#methods	%
application code	calling 'self changed'	74	11,95
UI code	calling both UI & application	524	35,99
	containing event handling code	32	2,20
	retrieving localisation information	128	8,79
	changing labels	50	5,11
	changing visibility	68	4,67

[2]The *MijnGeld* UI counts 30.737 lines of code. However, as 14.236 lines are concerned with the textual visualisation specifications (i.e. windowSpecs and resources), we omitted them from the line count.

The cases presented in the remainder of this chapter show how the UI of *MijnGeld* can be implemented with DEUCE. Recall that DEUCE allows to specify the several concerns in separation from one another and at different levels of abstraction. Furthermore the concerns are combined automatically into a final running application and also layout is computed and updated automatically. With these cases we illustrate that a good separation of concerns and sufficient support, allows developers to reuse UI specifications as well as to evolve parts of the specifications without affecting the other concerns. Furthermore the underlying application does no longer contain manually added update statements to inform the UI of changes. The higher level specifications take away the burden of the developers to have to know about low level implementation details that are specific for, in this case, VisualWorks Smalltalk user interfaces.

6.2 Reusable User Interface Specifications

The finance application *MijnGeld* offers the functionality to keep track of transaction statements of an account. An example of such a statement is withdrawing a certain amount of money from the account. Figure 6.1 shows the *MijnGeld* UI for this functionality. It lists the transactions that were made with the selected bank account and allows to add new transactions or to edit or delete transactions. Three types of transactions are supported, being withdrawals, deposits and transfers. For each of these types there is a dedicated input window that is similar to the others but with small differences nevertheless.

6.2.1 Implementation in VisualWorks Smalltalk

One way to implement the *MijnGeld* transaction windows in Smalltalk is by creating separate `windowSpecs` (i.e. Smalltalks window visualisation specification) for each of the transaction types. This would result in three different `windowSpecs` as shown in Figure 6.2. For each of the transaction types VisualWorks Smalltalk's UI Painter is used to draw the window visualisation and to generate a `windowSpec` out if. This `windowSpec` is a textual description of the visualisation and is in VisualWorks Smalltalk usually not edited manually but through the UI Painter.

A standard approach to introduce this kind of variability in Smalltalk UIs, is to create one `windowSpec` with the UIs visualisation and add code to adapt the window with the necessary variations. Hence, the developer of *MijnGeld* has opted for this appraoch and provides for the differences between the various transactions types programmatically. The following code for instance contains the code that changes to the transaction form for transfers. For clarification purposes we simplified this code by removing the lookup of labels in a message catalogue which is used for adapting the UI to localisation policies.

```
uiForTransfer
   builder isNil
   ifFalse:
      [(self wrapperAt: #splitButton) beInvisible.
```

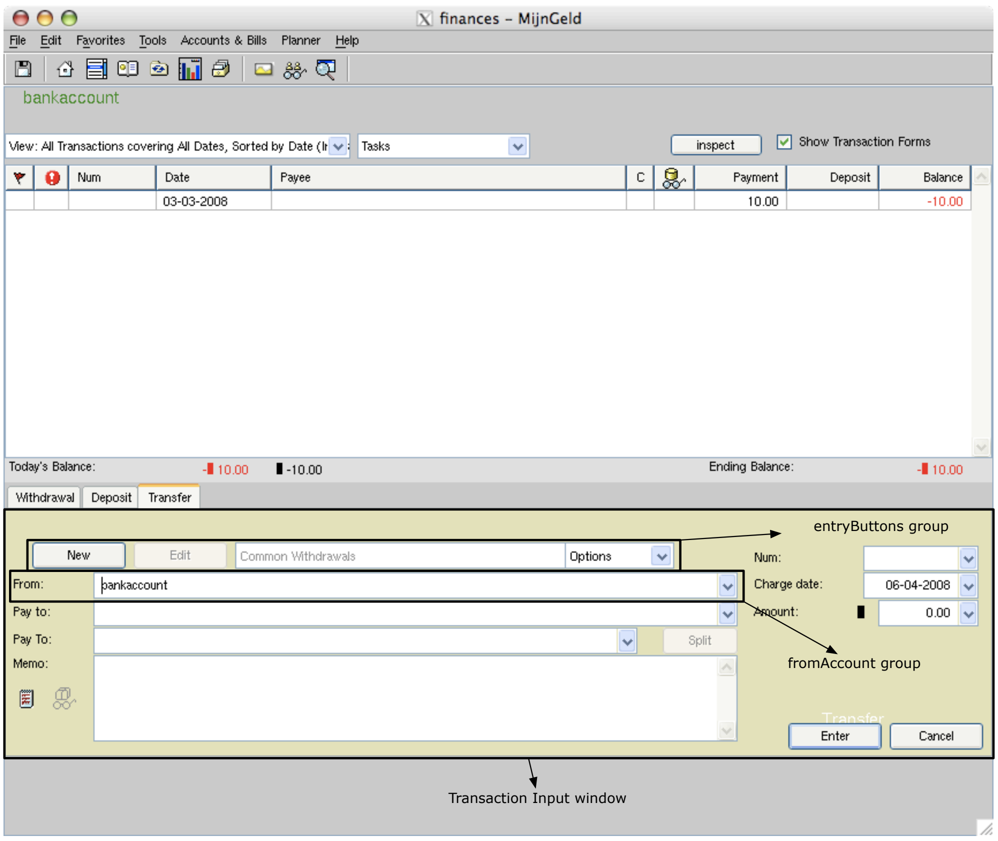
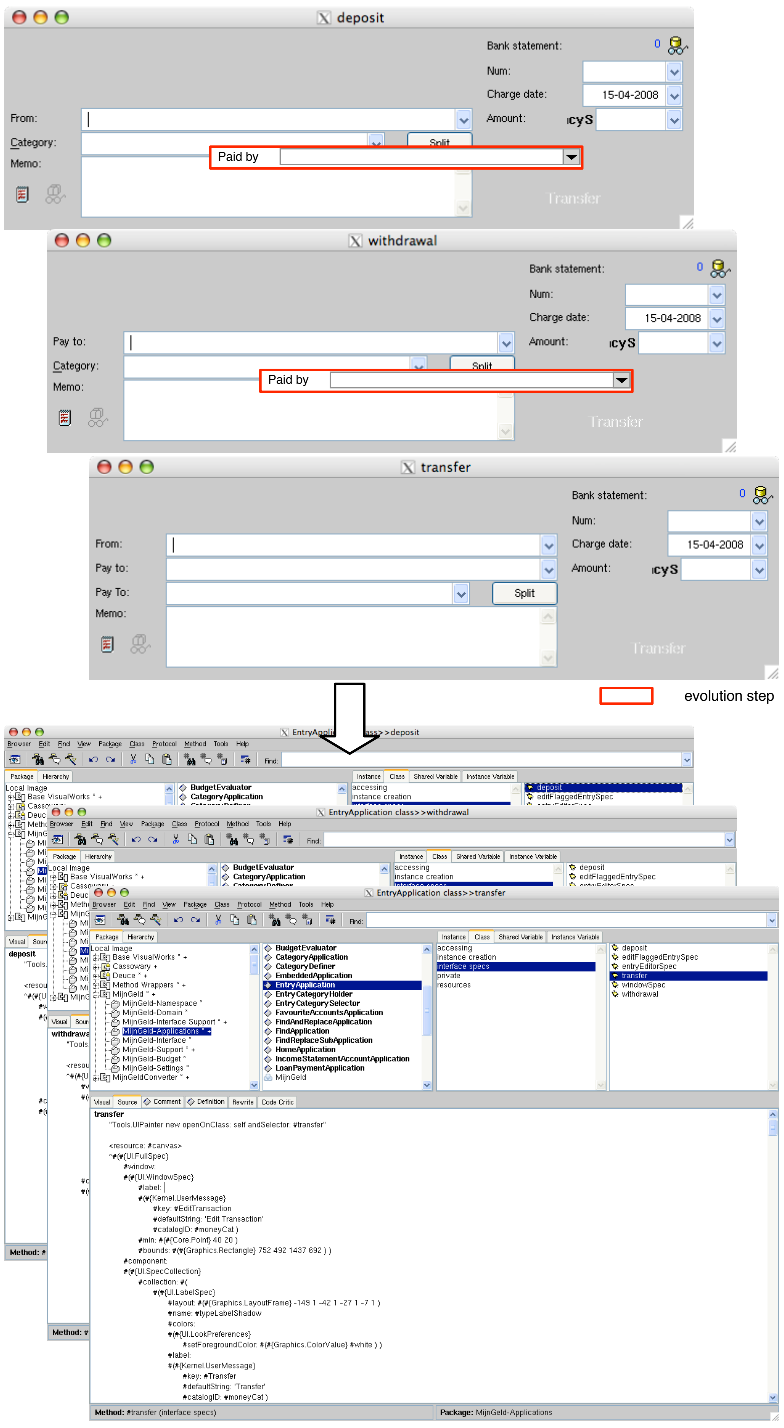


Figure 6.1: The bank statements window in *MijnGeld*



```
transfer
	"Tools.UIPainter new openOnClass: self andSelector: #transfer"

	<resource: #canvas>
	^#(#{UI.FullSpec}
		#window:
		#(#{UI.WindowSpec}
			#label:
			#(#{Kernel.UserMessage}
				#key: #EditTransaction
				#defaultString: 'Edit Transaction'
				#catalogID: #moneyCat )
			#min: #(#{Core.Point} 40 20 )
			#bounds: #(#{Graphics.Rectangle} 752 492 1437 692 ) )
		#component:
		#(#{UI.SpecCollection}
			#collection: #(
				#(#{UI.LabelSpec}
					#layout: #(#{Graphics.LayoutFrame} -149 1 -42 1 -27 1 -7 1 )
					#name: #typeLabelShadow
					#colors:
					#(#{UI.LookPreferences}
						#setForegroundColor: #(#{Graphics.ColorValue} #white ) )
					#label:
					#(#{Kernel.UserMessage}
						#key: #Transfer
						#defaultString: 'Transfer'
						#catalogID: #moneyCat )
```

Method: #transfer (interface specs) **Package:** MijnGeld-Applications

Figure 6.2: Transaction windows in *MijnGeld* and their corresponding windowSpecs

```
5        (self wrapperAt: #fromLabel) beVisible.
6        (self wrapperAt: #fromAccountText) beVisible.
7        (self wrapperAt: #fromAccountCombo) beVisible.
8        (self widgetAt: #fromAccountCombo) editor
9              helpText: 'Name of the account from which a booking is made'.
10       (self widgetAt: #toAccountCombo) editor
11              helpText: 'The account to which the entry is made'.
12       (self widgetAt: #categoryCombo) editor
13              helpText: 'The party involved in this transfer'.
14       (self widgetAt: #toLabel)
15              labelString: 'To:'.
16       (self widgetAt: #categoryLabel)
17              labelString: 'Pay to:'.
18       (self widgetAt: #typeLabel)
19              labelString: 'Transfer'.
20       (self widgetAt: #typeLabelShadow)
21              labelString: 'Transfer']
```

Lines 4–7 make certain components visible as they may have been turned invisible in other transaction windows but are of need for the transfer UI. The transfer transaction has two `'pay to'` fields of which one is used to record the account to which the transaction should be made and the other to keep track of whom the payment was made to. Therefore the category field, used in the other transaction types, is changed to act as a 'pay to' field on line 12 and line 16. However, this reuse has consequences on the rest of the UI application as the programmer has to keep in mind that this field is used for two different purposes depending on the transaction type. This is illustrated by the following code snippet which was taken from the *MijnGeld* implementation. Note that this is a representative example of how such variations are programmed in Smalltalk.

```
self category value isNil
   ifFalse:
      [(newEntry isTransfer and: [self isTransfer])
         ifTrue:  [newTransaction payee: self category value]
         ifFalse: [newTransaction category:
              (self category value addToSystem: self system
                                   under: self categoryType)]].
```

Furthermore, if decided to add a category field to the transfer transactions, the entire application has to be revised.

6.2.2 Evolving the Smalltalk Implementation

Assume the *MijnGeld* application to be evolved such that some types of transactions now include a 'paid by' to keep track of whom (e.g. of a family) made a certain payment, as illustrated in Figure 6.2. The field is to be positioned in-between the category and the memo field. Using the first implementation approach where each transaction type has its own visualisation, this requires the transaction visualisations to be updated manually. The new field has to be added and the other components need to be repositioned, and this operation is to be repeated for each of the `windowSpecs` for which this change is applicable. Using the second implementation approach where one `windowSpec` is changed programmatically, the code needs to be revised such that the deposit and withdrawal transaction take the new field into account. In the transaction window the field is made invisible, but remark that this would leave a gap in-between the ‘pay to’ (which is the category field) and the memo field.

6.2.3 Implementation in DEUCE

When specifying the transaction UIs with DEUCE, the common parts of the windows are specified in separate groups, which then can be reused by the different transaction types UI specifications. The following rules and facts specify the `fromAccount` group (see Figure 6.1) which is part of both the transfer and withdrawal window (lines 1–2).

```
1  containsComponent(fromAccountGroup,fromAccountLabel).
2  containsComponent(fromAccountGroup,fromAccountText).
3  containsComponent(fromAccountGroup,fromAccountCombo).

4  label(fromAccountLabel).
5  inputField(fromAccountText).
6  comboBox(fromAccountCombo).

7  minimumHeight(fromAccountLabel,23).
8  minimumHeight(fromAccountText,23).
9  minimumHeight(fromAccountCombo,23).
10 minimumWidth(fromAccountLabel,40).
11 minimumWidth(fromAccountText,400).
12 minimumWidth(fromAccountCombo,400)

13 colocate(fromAccountCombo,fromAccountText).
14 leftOf(fromAccountLabel,fromAccountCombo).
```

Lines 1–3 specify the components that are part of the `fromAccount` group. Lines 4–6 denote the type of components that are used, and lines 7–12 specify the height and width for these components. The layout positioning for the `fromAccount` group is declared on lines 13–14.

By grouping UI specifications, these groups can be reused in several of the transaction types windows. The statement on line 1 in the code snippet below is part of the UI specification for the deposit transaction and indicates that the fromAccountGroup is a ‘component’ (actually a group of components) being used in the deposit transaction UI. The statement on line 2 specifies the same for the transfer transaction.

```
containsComponent(depositTransaction,fromAccountGroup).
containsComponent(transferTransaction,fromAccountGroup)
```

Recall from Chapter 5 that the logic layers in SOUL can be used to create separate sets of specification facts and rules which can be loaded and unloaded from the runtime memory. For each of the groups above such a layer can be used to separate its specification from the other groups and reuse the entire specification in different UIs. For instance, the entire visualisation of the `fromAccountGroup` is reused in the several transaction windows such that the layout relations of the components within this group are the same for all transaction windows. It is no longer necessary to change components programmatically to make them act as a different component as was the case in the original *MijnGeld* implementation. With DEUCE the transfer window contains the `toAccount` group and the `payTo` group, but not the `category` group. The withdrawal window contains the `toAccount` group and the `category` group, but not the `payTo` group. This is specified in each of the UI specifications respectively as follows:

```
containsComponent(transferTransaction,toAccountGroup).
containsComponent(transferTransaction,payToGroup)

containsComponent(withdrawalTransaction,toAccountGroup).
containsComponent(withdrawalTransaction,categoryGroup)
```

When in future evolutions the `category` group should also become part of the transfer window, this is solved by adding the specification

```
containsComponent(transferTransaction,categoryGroup)
```

6.2.4 Evolving the DEUCE Specification

As groups of components are easily reused within DEUCE, we have specified the transaction windows as a whole, including the `entry buttons` group (see Figure 6.1). This group is reused in all windows, so that when adding an additional `search` button it is sufficient to add the facts that declare the search button to the UI specification of the entry buttons group. Note that in *MijnGeld* this would require all visualisation windows to be updated.

```
containsComponent(entryGroup,searchButton).
button(searchButton).

text(searchButton,search).
minimumWidth(searchButton,60).
minimumHeight(searchButton,20).
```

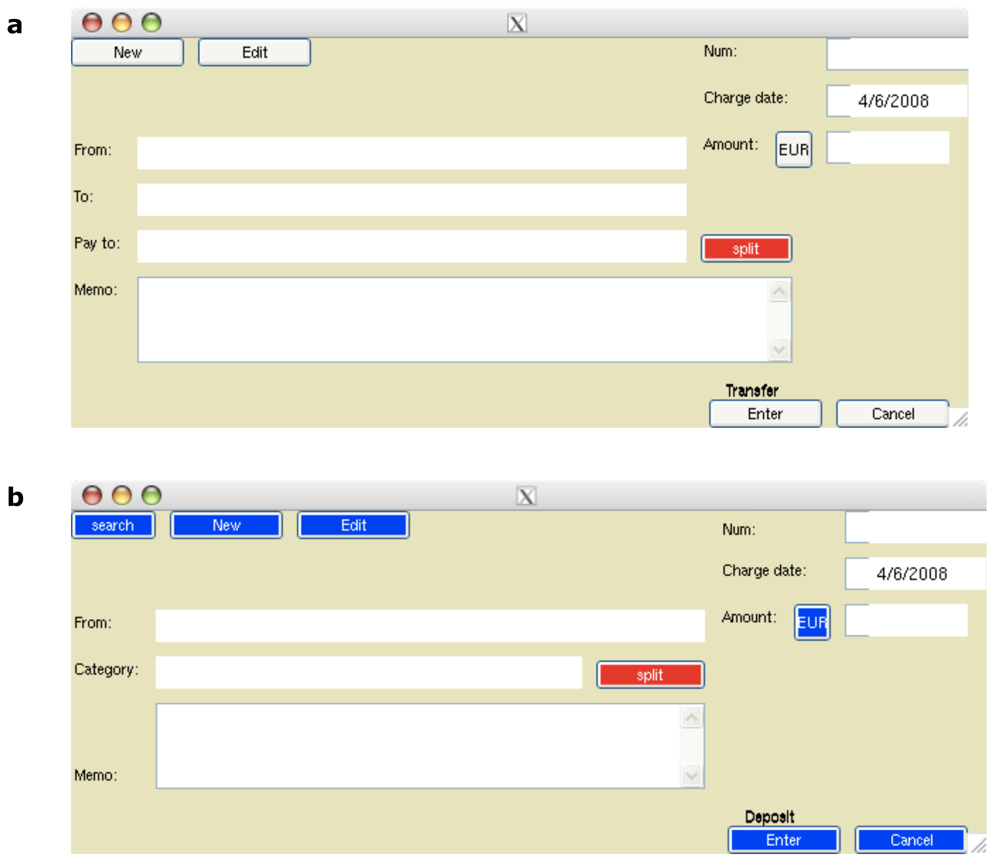


Figure 6.3: Transaction windows with DEUCE for: a) transfers b) deposits

```
leftOf(searchButton,newEntryButton).
topAlign(searchButton,newEntryButton)
```

With this extra specification from now on all windows using the `entryGroup` will display the search button.

In a similar way, the properties of components can be changed. For example, imagine we want the `split` button to be coloured red in all transaction windows. By adding the following property to the top layer specification (which is valid for all transaction types), the split button will automatically be coloured red in all windows. This is shown in Figure 6.3 where both the transfer and the deposit window now have a red split button.

```
foregroundColour(splitButton,white).
backgroundColour(splitButton,red).
```

Properties of components can also be changed for one particular type of transaction window by adding the specification to the set of facts for that window. For instance the following specification changes the colours of all buttons in the deposit window to blue. Note that the split button remains red and the buttons of the transfer window do not change.

```
backgroundColour(?comp,blue) if
   button(?comp).
foregroundColour(?comp,white) if
   button(?comp)
```

These rules specify that the `background` colour of a component that is of type `button` has to be blue and the `foreground` colour white.

Note that in VisualWorks Smalltalk the developers have to provide the code for the property changes themselves which requires knowledge on how to do this. For example in *MijnGeld* the label of a component is changed at 50 locations spread throughout the `applicationModels` code, but sometimes this is done by sending the message `label:` to the widget and other times by sending it `labelString:` or `setLabel:`. Furthermore the developers need to access the widget by sending the `widgetAt:` message to the UI instance. In DEUCE the details of how to access a widget and change its properties are hidden by the high-level UI specifications and mappings from high-level to low-level entities.

6.2.5 Reusing Visualisation Logic

Several parts of the UI concerns specifications can be reused for different UIs. For example in *MijnGeld* the developer used VisualWorks Smalltalk's support for localising the UI. To do so a 'message catalogue' is provided for each supported language. The text shown in labels, buttons, etc. is retrieved from this message catalogue. For instance for setting the transfer label in the transaction window for transfers, this is done as follows:

```
(self widgetAt: #typeLabel) labelString: (#Transfer << #moneyCat)asString
```

In this example the text for `Transfer` is looked up in the `moneyCat` message catalogue. In the English catalogue it maps onto 'Transfer' and in the Dutch catalogue it maps onto 'overboeking'. The disadvantage of using the message catalogue in VisualWorks Smalltalk is that all the code in the `applicationModels` that change the text of a label or button has to make the call to the catalogue. Hence these localisation statements are spread all throughout the *MijnGeld* implementation. Currently *MijnGeld* makes the lookup call in 8,79% of the methods in the `applicationModels`.

Additionally, applying localisation might result in labels and buttons to have to resize in order to show the entire text. For instance the Dutch word 'annuleren' is longer than 'cancel' so the cancel button bearing this label should be larger. Furthermore these resizes can result in a change in layout when for instance components are not to overlap and have to be aligned. Such problems are not taken care of by the VisualWorks Smalltalk support for localisation.

Using an approach like DEUCE does overcome these problems. First of all message catalogues are described declaratively, for instance as follows:

```
Layer for English
text(cancelButton,['Cancel'])

Layer for Dutch
text(cancelButton,['annuleren'])
```

Secondly, all the localisation for a certain language will be concentrated in the layer for that language. By adding these localisation layers to UI specifications, the localisation is automatically applied for the entire UI. There is no need of repeating the localisation statement, nor its lookup in the message catalogue, throughout the rest of the UI specification. Thirdly, as DEUCE calculates the layout based upon high-level layout relations, layouts are automatically adapted to the size of the new buttons and labels. For example, as the following layout relation states that the cancel button should be positioned to the right of the enter button but does not specify its actual position, its size is taken into account by the constraint solver upon calculating its actual position based on the layout relation.

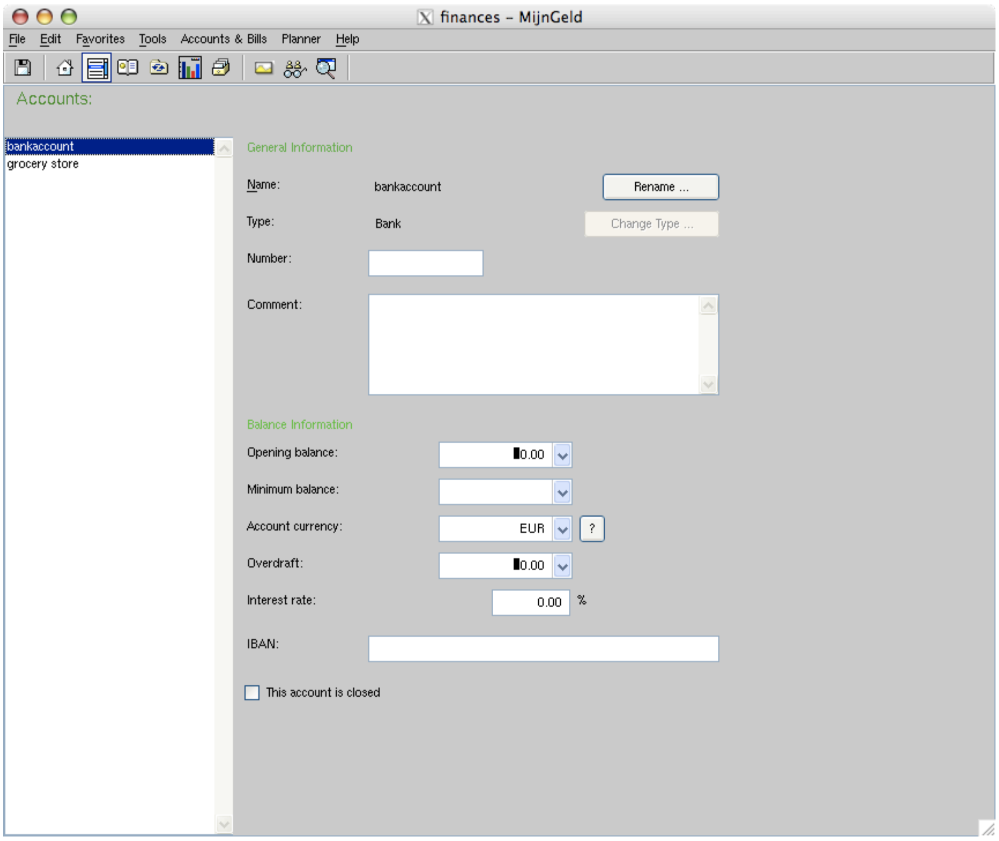
```
leftOf(enterButton, cancelButton)
```

The localisation layers can be reused for the different UIs in *MijnGeld* such that text is automatically adapted upon changing language. Similarly, layers with (partial) layout specifications can be reused such that company guidelines are enforced automatically. For example such guidelines specify font and colour properties, or logo's to be shown at certain places in the UI, etc. By using the layer for all UIs for a company, its guidelines ensure a consistent UI visualisation throughout all their applications. Also, if an application is used by different companies, each with their own guidelines, the particular UI for each of the companies is configured for that particular company.

6.3 Evolving the Presentation Logic Concern

When evolving the UI visualisation or its behaviour, the presentation logic will be affected. For example changing the way a button looks will affect the specification of that button. Changing the behaviour of the button such that it responds to clicking, double clicking and hoovering, also requires the presentation logic to be updated.

Figure 6.4 shows the *MijnGeld* window in which a user can change the details of a bank account, such as its account number and balance information. The underlying application model retrieves the information from the UI to update the application data. Providing a new visualisation strategy for this window, for instance to spread the several information groups over several windows, requires the application model to be



Accounts:

bankaccount
grocery store

General Information

Name: bankaccount

Type: Bank

Number:

Comment:

Balance Information

Opening balance: 0.00

Minimum balance:

Account currency: EUR

Overdraft: 0.00

Interest rate: 0.00 %

IBAN:

☐ This account is closed

Figure 6.4: Bank account details window in *MijnGeld*

updated. The new visualisations have to be provided, in the form of `windowSpecs`. The application model code has to be revised such that the right window is accessed when retrieving a widget's value for updating the application data.

6.3.1 Implementing the New Visualisation Logic With DEUCE

In Figure 6.5a the same interface is shown as it is generated by DEUCE.

In this specification the different groups of components are displayed one after the other by using the oneColumn rule:

```
containsComponent(bankAccount,generalInfo).
containsComponent(bankAccount,additionalInfo).
containsComponent(bankAccount,balanceInfo).
containsComponent(bankAccount,accountInfo).

oneColumn(<generalInfo,additionalInfo,balanceInfo,accountInfo>)
```

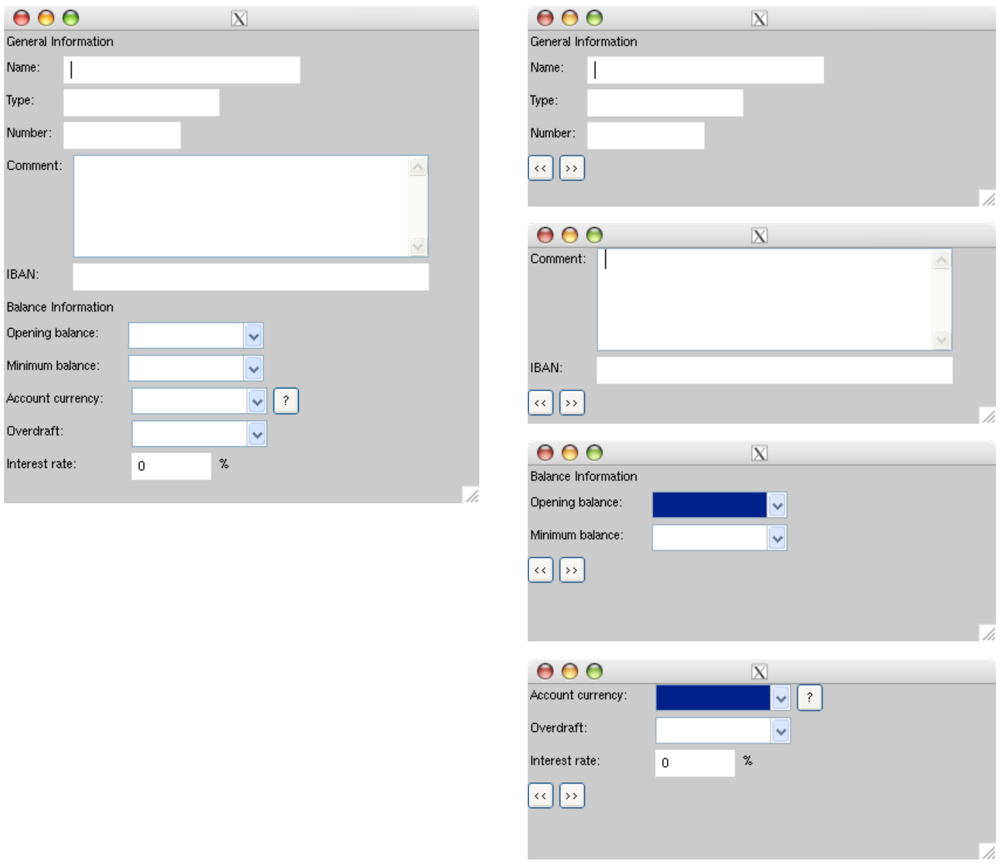


Figure 6.5: Bank account details window with DEUCE

When providing an alternative layout view for this UI as shown in Figure 6.5b, in the existing implementation of *MijnGeld* this requires providing four new window specifications. Once more making changes the the UI visualisation in this implementation, for instance by changing the properties of the iban field, requires changes to be applied to all different views by the developer manually. With DEUCE it is possible to provide for this alternative layout view by adding the following specifications.

```
1  window(generalInfo).
2  window(additionalInfo).
3  window(balanceInfo).
4  window(accountInfo).

5  group(flowButtons,<previous,next>).
6  button(previous).
7  button(next).
8  text(previous,['<<']).
9  text(next,['>>']).

10 sameRow(<previous,next>).
11 above(?x,flowButtons) if
12    group(?x,?comps)
```

Lines 1–4 specify that the component groups are actually a window, such that DEUCE will generate a separate UI window for each of these. Lines 5–9 specify the next and the previous button which are to be added to each of the windows of the UI, in order to go the next or previous window. These buttons are positioned next to one another (line 10) and below the other components in the group (line 11).

6.3.2 Implementing the New Behavioural Logic With DEUCE

The visualisation alternative has been specified, and additional flow buttons are added. However the behavioural logic for these buttons has to be specified as well such that clicking these buttons results in opening the next or previous window in the control flow. To do so, the following is added to the behaviour logic of the Bank account UI specification.

```
1  linkUIEventToQuery(previousButtonClicked,flowButton(?ui,previous)).
2  linkUIEventToQuery(nextButtonClicked,flowButton(?ui,next)).

3  UIEvent(nextButtonClicked,next,click).
4  UIEvent(previousButtonClicked,previous,click).

5  next(Window1,Window2).
6  next(Window2,Window3).
7  next(Window3,Window4).
8  previous(Window4,Window3).
```

```
9   previous(Window3,Window2).
10  previous(Window2,Window1).

11  flowButton(?ui,previous) if
12     currentWindow(?ui, ?window),
13     previous(?window,?newWindow),
14     changeCurrentWindowTo(?ui,?window,?newWindow).

15  flowButton(?ui,next) if
16     currentWindow(?ui, ?window),
17     next(?window,?newWindow),
18     changeCurrentWindowTo(?ui,?window,?newWindow).
```

The statements on lines 1–4 specify that clicking the previous or next button results in triggering the flowButton query. Recall that these statements are transformed by DEUCE behind the scenes into event handlers on the corresponding button. In the current implementation of *MijnGeld* the developer provides these handlers manually. On the one hand by using the UI Painter Tool to link a component to a model method for which the code is provided in the `applicationModel`. On the other hand by writing event instalment statements. The latter is less common but nevertheless necessary when using events that differ from the standard model, for instance when the event for a button is not the click event. In *MijnGeld* 2,2% of the methods makes such an installation call on top of what was specified as the default model for the UI widgets.

Lines 5–10 are facts being used in the `flowButton` query to know what window to open as the previous or the next window.

6.4 Evolving the Application Logic Concern

Introducing new behaviour in the application that is to be reflected in the UI, requires the application logic to evolve. However, it does not necessarily affect the other UI concerns. For example, Figure 6.6 is a snapshot of the *MijnGeld* user interface that is used to keep track of payees being used in the finance application. The list of payees can be adapted with this interface. Introducing a new implementation for the payees objects, results in new application logic to call upon these new objects. There is no need to change the presentation logic (as the UI remains the same), nor the connection logic (as the link between presentation and application logic remains the same).

6.4.1 Implementation in VisualWorks Smalltalk

The payees window provides a view on the `parties` database of the `MijnGeld.System` class. Therefore the window is defined for the application model `PartiesApplication` Class, which provides the actual links between the widget values and the contents of the `parties` database. The application model `PartiesApplication` contains the code to retrieve or update information in `parties` as well as the code to retrieve or update

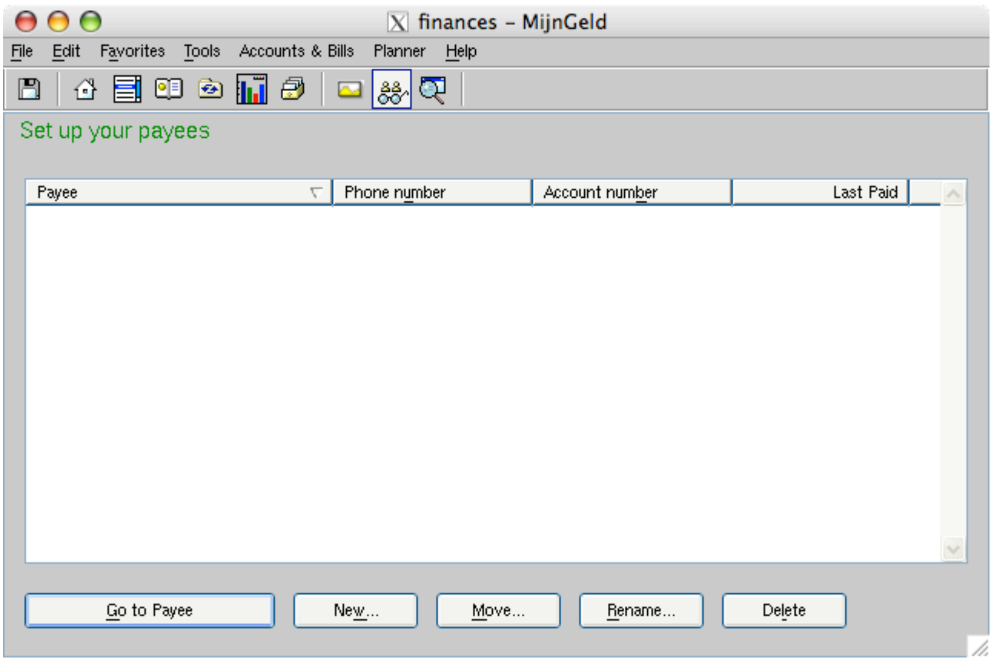
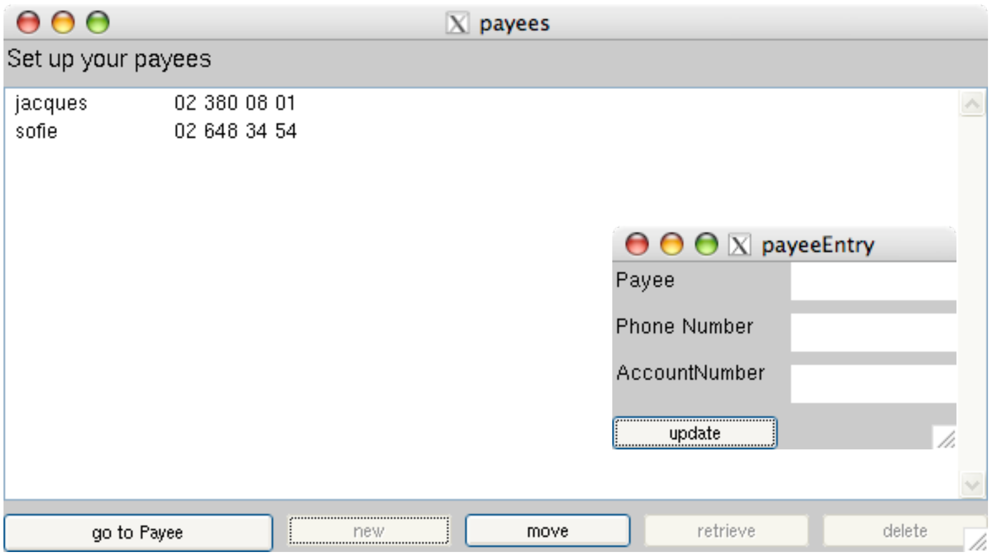


Figure 6.6: Payees window in *MijnGeld*

the values of the widgets being used in the UI. For example, the following method of PartiesApplication is used to delete a payee entry.

```
1  deleteParty
2     | toRemove index |
3     toRemove := self selectedRow value.
4     index := self payeesList selectionIndex - 1 max: 1.
5     self payeesList list remove: toRemove.
6     self system removeParty: toRemove.
7     self payeesList selectionIndex: index
```

On line 6 the entry is deleted from the `parties` database. On line 5 it is deleted from the list of payees, which functions as the value for the widget displaying this list, while the other lines of code are used to update the index pointer of that same list. As such the method triggers UI logic, apart from the call on line 6 which triggers the underlying application. 36% of the methods implemented in the `applicationModels` of *MijnGeld* (and the same accounts for other Smalltalk systems), call upon both the UI and the application.



```
jacques    02 380 08 01
sofie      02 648 34 54
```

Figure 6.7: Payees window with DEUCE

6.4.2 Evolving the Smalltalk Implementation

Let us assume that the *MijnGeld* application is evolved such that it uses a new kind of `parties` database which is accessed through different methods. For instance, removing an entry is now done by sending the `delete:` method. Hence, the method call on line 6 has to be changed. It is up to the developers to know where the method is used and to understand what lines change and which lines remain the same.

6.4.3 Implementation in DEUCE

We specified the payees UI with DEUCE, which resulted in the UI shown in Figure 6.7. We omit the specification of the visualisation logic as it is similar to the examples shown before. In what follows we give details on the other concerns with respect to the delete functionality.

The **behavioural logic** is shown first.

```
UIEvent(deleteButtonClicked,deleteButton,click).

linkUIEventToQuery(deleteButtonClicked,deleteEntry(?ui)).

deleteEntry(?ui) if
   deletePayee(?ui).

deleteEntry(?ui) if
   disable(updateButton, ?ui).
```

```
7   deleteEntry(?ui) if
8      disable(retrieveButton, ?ui).

9   deleteEntry(?ui) if
10     disable(deleteButton, ?ui).

11  deleteEntry(?ui) if
12     enable(newButton, ?ui)
```

The `UIEvent` fact on line 1 specifies that the clicking to the component `deleteButton` will result in triggering the `deleteButtonClicked` event. This event is linked to the `deleteEntry(?ui)` query on line 2 such that upon the `deleteButtonClicked` event this query is launched. The specification of the `deleteEntry` query is specified on lines 3–12. When triggered, it launches the `deletePayee` query which is explained further on as it is part of the connection logic. Furthermore the query `deleteEntry` will change the properties of certain other buttons, namely by enabling and disabling them. Note that this enablement and disablement of buttons is not present in the original *MijnGeld* application as it would complicate its current implementation. However in the DEUCE version of the UI it is added with little overhead.

The **application logic** with respect to the delete functionality is the following:

```
1   applicationEvent(payeeRemoved,payeeDatabase,#removeParty:).

2   role(payeeDatabase,[Sepher.MijnGeld.System]).

3   linkApplicationEventToQuery(payeeRemoved,payeeRemoved(?ui)).
```

The `applicationEvent` fact on line 1 specifies that sending the message `removeParty:` to the `payeeDatabase` (which is represented by the class `[Sepher.MijnGeld.System]` as indicated on line 3) triggers the application event `payeeRemoved`. This event maps to the query `payeeRemoved(?ui)` which will be launched when the event is triggered. The specification of the query itself is part of the connection logic, as will be shown further on. As we explained in Chapter 5, the application events are transformed by DEUCE into method wrappers automatically. By doing so developers no longer have to browse the underlying application to add the necessary observer statements. Currently in VisualWorks Smalltalk this is required and these statements contaminate the application code. In the *MijnGeld* implementation for example these statements were added to 11,95% of the methods in the core application.

Additionally, the application logic also contains the code to actually update the application. For instance, the following is used to remove an entry from the underlying application.

```
1   removePayee(?appl,?payee) if
2      [?appl removeParty: ?payee. true]
```

Upon launching this query, the underlying application is called to actually remove the selected entry.

The **connection logic** provides the link between UI and application events and UI and application actions. With respect to the delete button, we have seen that the UI behavioural logic and the application logic trigger the connection events `deletePayee` and `payeeRemoved`. These are specified as follows:

```
deletePayee(?ui) if
   application(?ui,?appl),
   removePayee(?appl,?payee).

payeeRemoved(?ui) if
   updatePayeeDisplay(?ui)
```

The first query calls the `removePayee` application action. The second query calls the `updatePayeeDisplay` UI action.

6.4.4 Evolving the DEUCE Specification

When evolving the application in the same way as before by providing a new kind of parties database and new methods to access this database, this has consequences for the application logic only. The application logic provides the link between the underlying application and the UI concerns. The other UI concerns, namely the connection and presentation logic, access the application through the abstractions of the application logic. Changing the underlying application hence requires to update the body of these abstraction queries. For example, for the delete functionality, the application logic changes are:

```
applicationEvent(payeeRemoved,payeeDatabase,#deleteParty:).

removePayee(?appl,?payee) if
   [?appl deleteParty: ?payee. true]
```

The other concerns are not affected by this application evolution, as should be the case.

6.5 User Interface Events Trigger Connection Logic

The connection logic of a user interface is the concern where UI and application actions are linked with each another such that upon a UI or application event, these actions are triggered. As such it provides for the glue between the presentation logic and the application logic concern. Figure 6.8 depicts an example where a UI event triggers connection logic which results in calling both application and UI actions. Consider the bank account details window of the *MijnGeld* application. It contains a field for the input of the bank account number. When the user changes the number, its input is checked for validity. Depending on its validity, the field is coloured differently such that the user gets visual feedback on his input. If the account number matches the pattern for account numbers, it is partially valid and the field is coloured orange. If additionally its sequence also succeeds the validation checksum, the field is coloured green. Otherwise the input number is invalid and coloured red.

6.5.1 Implementation in VisualWorks Smalltalk

The functionality of the example shown in Figure 6.8 is not implemented in the original *MijnGeld* application but as it shows how context knowledge (i.e. application data) influences the UI and its link with the application, we will show next how it can be implemented in Smalltalk. As this functionality is concerned with the user interface, it is implemented in the `applicationModel` class of the bank account details window. As this class combines all UI concerns, namely presentation, application and connection logic, the code tangling between these concerns is omnipresent and complicates adding functionality as illustrated by the example.

First of all the developers need to install the event handler on the `numberInputField` such that upon changing it, the `applicationModel` is called. Typically this code is put in the `postBuildWith:` method of the `applicationModel` which is executed after the runtime Smalltalk UI window has been built. The following installs an event handler on the `numberInputField` component which sends the `accountNumberInputed` message to the `applicationModel` instance (bound to `self`) upon the `changed` event.

```
postBuildWith: aBuilder
   super postBuildWith: aBuilder.
   aBuilder componentAt: #numberInputField widget
               when: #changed
               send: #accountNumberInputed
               to: self
```

The `accountNumberInputed` method is responsible for updating the UI visualisation. As shown below it checks the validity of the account number being inputted and decides what UI update method to call. Also these update methods are to be implemented by the developers in the `applicationModel` class.

```
accountNumberInputed
   |input|
   input := self numberInput value.

   (self isValidAccountNumberPattern: input)
      ifTrue:  [
         (self isValidAccountNumberSequence: input)
            ifTrue:  [self validInput]
            ifFalse: [self partiallyValidInput]]
      ifFalse: [self invalidInput]
```

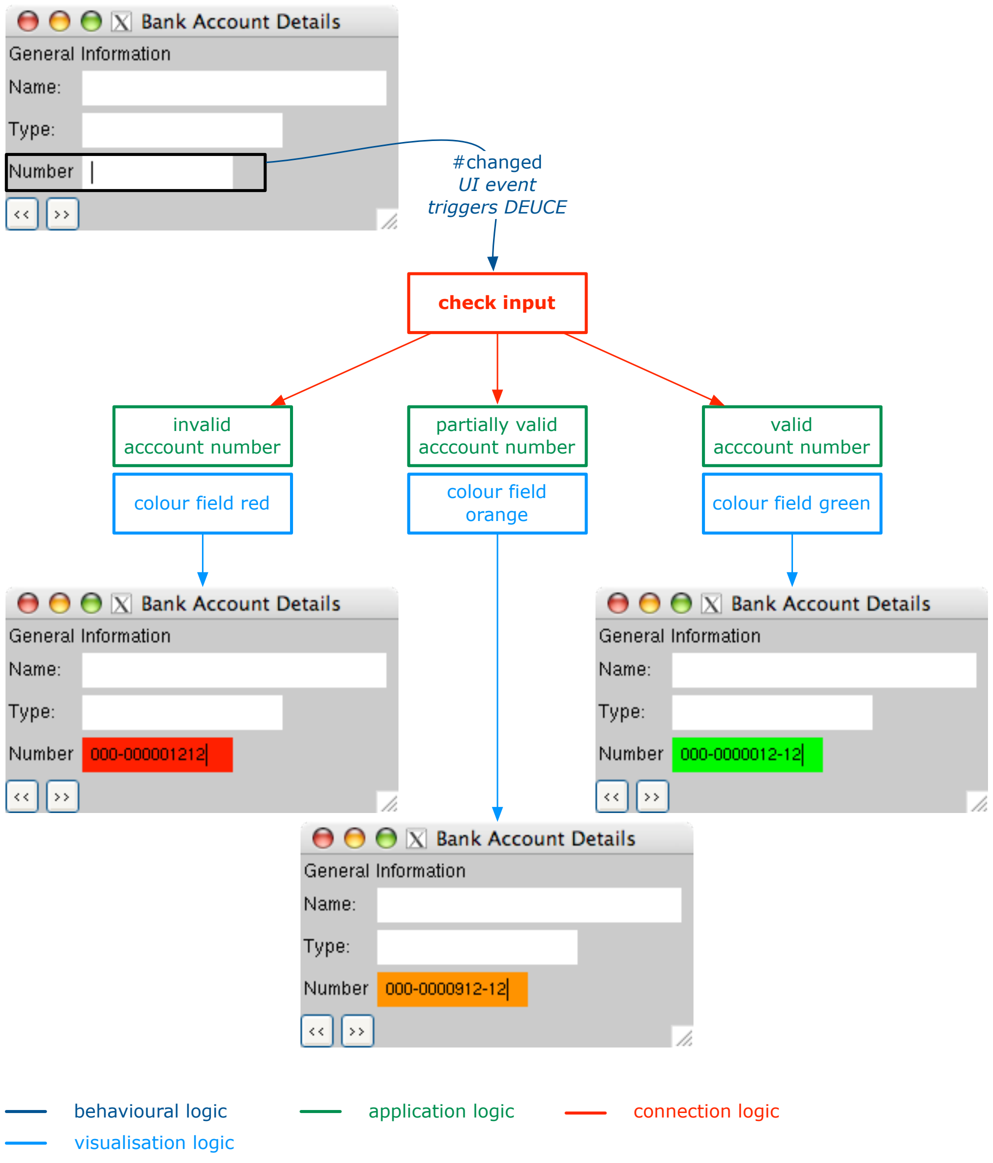


Figure 6.8: User interface events trigger connection logic

Updating the UI visualisation, for which the code is shown below, requires developers to know how to retrieve the component (line 3) and how to change its background colour (lines 4–6).

```
1  validInput
2     |widget prefs|
3     widget := self builder componentAt: #numberInputField.
4     prefs := widget lookPreferences.
5     prefs setBackgroundColor: (ColorValue green).
6     widget  lookPreferences: prefs

7  partiallyValidInput
8     ...
9  invalidInput
10    ...
```

6.5.2 Implementation in DEUCE

Implementing the example in DEUCE separates the knowledge put in the `applicationModel` class into the several concerns. Furthermore DEUCE provides high-level abstractions such that developers are no longer concerned with the actual implementations of how to add event handlers or update component properties. Note that this does assume the abstractions and 'mappings' from high-level to low-level entities to be provided. However, libraries with this knowledge can be reused if the solution for separation of concerns provides the necessary infrastructure to do so.

The behavioural logic of the example is shown below. Line 1 specifies that the changing the account number (`accountNumberChanged`) is linked to the changed event on the `numberInputField`. It is linked to the `checkAccountNumber` query on line 2. The rule `accountNumberInput` on line 3 is used to retrieve the value of the `numberInputField` component. The rule `valueForComponent` is part of the underlying rule set where is specified how to access an actual Smalltalk UI component.

```
1  UIEvent(accountNumberChanged,numberInputField,changed).

2  linkUIEventToQuery(accountNumberChanged,checkAccountNumber(?ui)).

3  accountNumberInput(?ui,?number) if
4     valueForComponent(numberInputField,?ui,?number)
```

The behavioural logic also contains how valid or invalid input is shown in the interface. The following code example shows how the queries `validInput`, `partialValidInput` and `invalidInput` are mapped onto changing the background colour of the `numberInputField`. These queries are triggered by the connection logic. When the way this validity is shown in the interface is evolved, these queries are updated in the behavioural logic while the connection logic will remain unchanged.

```
validInput(accountNumber,?ui) if
   backgroundColour(numberInputField,?ui,green).

partialValidInput(accountNumber,?ui) if
   backgroundColour(numberInputField,?ui,orange).

invalidInput(accountNumber,?ui) if
   backgroundColour(numberInputField,?ui,red)
```

The application logic specifies what it means for an account number to be valid as shown in the following code snippet. As *MijnGeld* does not provide this functionality, we added it directly in the application logic layer as a logic rule for illustration purposes. Nevertheless this functionality is business logic that should be provided by the underlying application. In that case the application logic will map the `validAccountNumberSequence` to a method call to the underlying application.

```
validAccountNumberSequence(?appl,?number) if
   [(((?number copyFrom:1 to: 3), (?number copyFrom: 5 to: 11)) asNumber
                \\ 97) == ((?number copyFrom:13 to: 14) asNumber)].

validAccountNumberPattern(?appl,?number) if
   ['###-#######-##' match: ?number]
```

Finally, the connection logic is specified in order to link the UI event that triggers the `checkAccountNumber` query to UI and application actions.

```
1   checkAccountNumber(?ui) if
2      application(?ui,?appl),
3      accountNumberInput(?ui,?number),
4      validAccountNumberSequence(?appl,?number),
5      validAccountNumberPattern(?appl,?number),!,
6      validInput(accountNumber,?ui).

7   checkAccountNumber(?ui) if
8      application(?ui,?appl),
9      accountNumberInput(?ui,?number),
10     validAccountNumberPattern(?appl,?number),!,
11     partialValidInput(accountNumber,?ui).

12  checkAccountNumber(?ui) if
13     invalidInput(accountNumber,?ui)
```

The `checkAccountNumber` rule has three alternatives. The first alternative on line 1 retrieves the input in the `numberInputField` (line 3), checks whether both the sequence (line 4) and pattern (line 5) are valid, and if so, updates the UI (line 6). The second alternative (line 7) checks for partially valid input and the last alternative (line 12) is triggered if the first two rules both failed and hence the input is invalid.

6.6 Application Events Trigger Connection Logic

In the previous example we have seen how UI events trigger a connection logic query, but also application events can do so. Let us reconsider the example of the bank account details window with added functionality as shown in Figure 6.9.

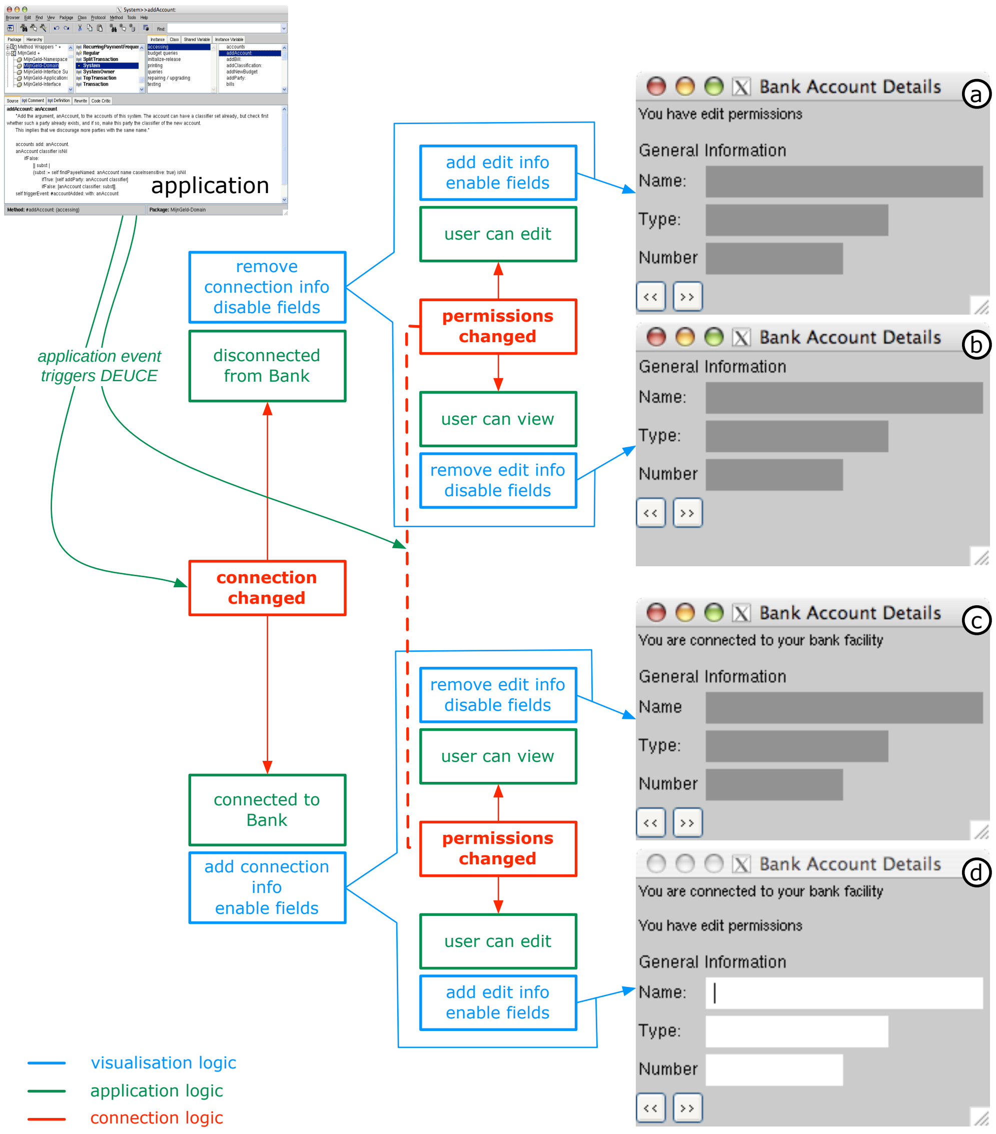
A first example assumes *MijnGeld* to be connected to an online banking facility. If there is a connection, an extra information label about the connection is added and the user is allowed to edit the bank details. Therefore the input components are enabled. If the connection is absent, the extra information label is removed. Furthermore the bank account details can be only be viewed by the user. This results in the input components being disabled. A second example assumes *MijnGeld* to be used by different users (for instance from the same family) where some users have edit permissions while others only have view permissions. If a user has edit permissions an edit information label is added and input fields are enabled. In the case of view permissions, the information label is removed and the input fields are disabled. However, as the first and second example are brought together in the same *MijnGeld* application, they influence each other. If a user has edit permissions but the application is not connected to the bank facility, the user has edit rights but can not change the details.

6.6.1 Implementation in VisualWorks Smalltalk

Implementing this example is VisualWorks Smalltalk is similar to the implementation of the previous example, apart from the fact that the observer mechanism will be triggered by the underlying application such that the `applicationModel` is called. In order for the observer mechanism to be triggered, developers need to add '`self changed`' statements to the application methods manually. For instance, if the UI is updated upon changing the connection status, the application code should look like:

```
connectionStatus: aBoolean
   connected := aBoolean.
   self changed
```

These 'self changed' messages are contaminating the underlying application code and have to be added by the developers manually. Upon the '`self changed`' message, all the UIs (i.e. views) that depend on the application are notified of the change and are responsible for updating themselves accordingly. Recall that the code for these updates is part of the `applicationModel` class, consisting of entangled presentation logic, application logic and connection logic. Hence, in addition to changing the underlying application, the developers provide the UI logic to update the UI windows in the `applicationModel`. As labels are added and removed at runtime and other components are repositioned, the



```
addAccount: anAccount
	"Add the argument, anAccount, to the accounts of this system. The account can have a classifier set already, but check first whether such a party already exists, and if so, make this party the classifier of the new account.
	This implies that we discourage more parties with the same name."

	accounts add: anAccount.
	anAccount classifier isNil
		ifFalse:
			[| subst |
			(subst := self findPayeeNamed: anAccount name caseInsensitive: true) isNil
				ifTrue: [self addParty: anAccount classifier]
				ifFalse: [anAccount classifier: subst]].
	self triggerEvent: #accountAdded: with: anAccount
```

Figure 6.9: Application events trigger connection logic

developers need to implement this behaviour. Although this code will be more complex, it is similar to the example from the Section 6.5.

6.6.2 Implementation in DEUCE

Connection Status changes

Implementing the first part of the example in which the UI is updated depending on the connection status of the application is done as follows. First of all the `connectionStatus:` method of the *MijnGeld* application is specified as being an application event, namely `connectChanged` (line 1). This application event will trigger the connection query `connectionStatusChanged` (line 2).

```
1 applicationEvent(connectChanged,mijnGeld,#connectionStatus:).
2 linkApplicationEventToQuery(connectChanged,connectionStatusChanged(?ui))
```

The connection query `connectionStatusChanged` calls upon two other connection queries that will make sure the extra information (i.e. label) is shown and the usage possibilities (namely editable or viewable) are applied.

```
1 connectionStatusChanged(?ui) if
2    extraInformation(?ui),
3    usagePossibilities(?ui)
```

Showing the extra information related to the connection status is specified by the following two rules:

```
1 extraInformation(?ui) if
2    application(?ui,?appl),
3    connected(?appl),
4    connectionInfo(?ui,connected).

5 extraInformation(?ui) if
6    application(?ui,?appl),
7    not(connected(?appl)),
8    connectionInfo(?ui,disconnected)
```

The application logic is queried to find out whether the underlying application is connected to its bank institution on lines 3 and 7. The behavioural logic is triggered to adapt accordingly on lines 4 and 8.

Similarly the usage possibilities of the UI are updated:

```
1 usagePossibilities(?ui) if
2    application(?ui,?appl),
3    connected(?appl),
4    editState(?ui).
```

```
usagePossibilities(?ui) if
   application(?ui,?appl),
   not(connected(?appl)),
   viewState(?ui)
```

The application logic for checking the connection status is specified below and shows that a message is sent to the underlying application in order to find out.

```
connected(?appl) if
   [?appl isConnected]
```

In the behavioural logic concern we specify how extra information is shown in the user interface and how the usage possibilities (`editState` and `viewState`) are carried through. For example if the connection information is to be shown for the connected status, the `onlineLabel` is added to the bank account UI (line 3), namely to its window responsible for showing the general information (line 2). After adding the new component, the window's layout is updated (line 4).

```
1  connectionInfo(?ui,connected) if
2     ?storage->window(?ui,generalInformationWindow,?window),
3     ?UISpecRules->addComponentTo(onlineLabel,?window),
4     ?UISpecRules->calculateLayout(?window).

5  connectionInfo(?ui,disconnected) if
6     ?storage->window(?ui,generalInformationWindow,?window),
7     ?UISpecRules->removeComponentFrom(onlineLabel,?window),
8     ?UISpecRules->calculateLayout(?window)
```

Applying the edit state on the UI is done by enabling the input components, while they are disabled for the view state.

```
editState(?ui) if  enable(nameInputField,?ui).
editState(?ui) if  enable(numberInputField,?ui).
editState(?ui) if  enable(typeInputField,?ui).

viewState(?ui) if  disable(nameInputField,?ui).
viewState(?ui) if  disable(typeInputField,?ui).
viewState(?ui) if  disable(numberInputField,?ui).
```

In this implementation we have listed all components that are to be enabled of disabled. An alternative specification is the following:

```
editState(?ui) if
   inputField(?component,?ui),
   enable(?component, ?ui).
```

```
viewState(?ui) if
   inputField(?component,?ui),
   disable(?component,?ui)
```

In this alternative specification all `inputFields` are enabled or disabled. When adding a new `inputField` component to the UI, the behavioural logic does not change as these rules (and the reasoning engine) assure that all `inputField` components are updated.

User Permissions changes

Changing the UI when the user permissions change is implemented similarly to the connection status change. An application event is specified for calling the `currentUser:` method in the *MijnGeld* application (line 1). The event is linked to the `userPermissionsChanged` query (line 2).

```
1  applicationEvent(userChanged,mijnGeld,#currentUser:).
2  linkApplicationEventToQuery(userChanged,userPermissionsChanged(?ui))
```

The connection query `userPermissionsChanged`, shown in the code snippet bellow, also calls upon the `extraInformation` (line 2) and the `usagePossibilities` (line 3) query, which in their turn will call upon the application and presentation logic.

```
1  userPermissionsChanged(?ui) if
2     extraInformation(?ui),
3     usagePossibilities(?ui).

4  extraInformation(?ui) if
5     application(?ui,?appl),
6     userPermissions(?appl,view),
7     permissionInfo(?ui,view).

8  extraInformation(?ui) if
9     application(?ui,?appl),
10    userPermissions(?appl,edit),
11    permissionInfo(?ui,edit).

12 usagePossibilities(?ui) if
13    application(?ui,?appl),
14    userPermissions(?appl,view),
15    viewState(?ui).

16 usagePossibilities(?ui) if
17    application(?ui,?appl),
18    userPermissions(?appl,edit),
19    editState(?ui)
```

The `editState` and `viewState` queries are defined in the behavioural logic and are the same rules as shown before. The userPermissions query is an application logic query and is specified as:

```
1  userPermissions(?appl,?permission) if
2     equals(?permission, [?appl getUserPermission])
```

As shown next, in similarity with the `connectionInfo` rule, the `permissionInfo` rule is part of the behavioural logic and adds (line 3) or removes (line 7) an information label to the `generalInformationWindow` (lines 2 and 6).

```
1  permissionInfo(?ui,edit) if
2     ?storage->window(?ui,generalInformationWindow,?window),
3     ?UISpecRules->addComponentTo(editLabel,?window),
4     ?UISpecRules->calculateLayout(?window).

5  permissionInfo(?ui,view) if
6     ?storage->window(?ui,generalInformationWindow,?window),
7     ?UISpecRules->removeComponentFrom(editLabel,?window),
8     ?UISpecRules->calculateLayout(?window)
```

Connection Status and User permissions combined

As shown in Figure 6.9, combining both context changes results in different UI behaviour. Although the application is connected to its financial institution, the users can not edit the information if they have no edit permissions (Figure 6.9c). Additionally if users have edit permissions but the application is not connected, the bank account details can not be edited. As both context changes influence each other, an extra rule has to be added to the connection logic such that their influence on each other is taken into account.

```
1  usagePossibilities(?ui) if
2     application(?ui,?appl),
3     userPermissions(?appl,edit),
4     not(connected(?appl)),
5     viewState(?ui)
```

This rule specifies that if a user has edit permissions (line 3) but the application is disconnected from the financial institution (line 4), the user interface does not allow to edit the details and hence is in its `viewState` (line 5). Note that the previous specifications of the `usagePossiblities` rule remain valid. Since the declarative reasoning mechanism will also execute these other alternatives for `usagePossiblities`, it assures that the UI is updated as needed.

6.7 Conclusion

In this chapter we have shown that the conceptual solution from Chapter 3 and its practical implementation DEUCE from Chapters 4 and 5 achieves the separation of concerns we envisioned, also at the implementation level of user interfaces and applications.

A first consequence of separating user interface concerns is the increase in reuse possibilities. In Section 6.2 we illustrate this by reusing groups of components in several UIs. As a bonus we showed how localisation statements, which typically are scattered throughout the entire UI code, can be centralised into one layer of specifications and be reused in several UIs. Furthermore the validation scenario in this section benefits from the strength of declarative programming for easily updating the properties of UI components.

In Section 6.3 the presentation logic concern was evolved by adding an additional presentation strategy for one of the *MijnGeld* windows such that the UI was split over several windows and new flow buttons were added to switch between these windows. This new strategy implied changes to both the visualisation logic and the behavioural logic respectively. The other concerns were not affected though.

Section 6.4 evolves the application logic concern in both the Smalltalk and DEUCE implementation of *MijnGeld*. Whereas the Smalltalk implementation this implied changes throughout the UI implementation (i.e. `applicationModel` classes), in DEUCE the concern does not affect the other UI concerns.

The scenarios in Section 6.5 and Section 6.6 focussed on evolving the connection logic concern. This concern can be triggered by both UI events and application events. As a result a connection logic query is launched. The connection logic is expressed at the logical level and benefits from the abstractions offered by DEUCE. These abstractions as well as support for the dynamic UI changes lack in the Smalltalk implementation, which hampers the implementation of both scenario's in Smalltalk. However, both examples illustrate how a good separation of concerns facilitates the implementation of advanced user interfaces.

Note that we have not validated the practicality of DEUCE, but rather how the various user interface concerns can be separated from one another such that a modularisation is accomplished. In order to validate the practicality and adoptability of DEUCE, an empirical study would need to be conducted. This implies having two sufficiently large groups of developers develop a similar software system. Whereas one group uses the contemporary approach, the other would need to use DEUCE. Additionally we believe a more mature DEUCE implementation (in contrast to this proof-of-concept implementation) is needed to do so. Nevertheless we are confident that DEUCE (or similar approaches) supports developers in writing application and user interface code. Various presentations about this work were given in an industrial context (e.g. for partners in the Advanced Media project and for partners from inno.com) and made it clear that the problem we tackled in this dissertation is indeed a problem that is perceived in practice. Furthermore we received positive feedback on our approach to tackle the problem.

Chapter 7

Conclusion and Future Work

The advent of new programming paradigms and development methods, such as ambient intelligence, context-sensitive systems and agile development, give rise to numerous new software challenges. One recurring challenge that developers face in all of these is achieving a clean separation of concerns in user interfaces. Although the principle of separation of concerns is not new, it is poorly supported by current software development environments. Nonetheless, adding or evolving concerns that are scattered and entangled has a big impact on the application's code. A scattered concern implies an invasive change to different modules. Tangled code impedes comprehension and future evolution. Because of code scattering and tangling, making changes to one concern might break the functionality of another one.

In this dissertation we presented a solution to achieve the modularisation of user interface code from the application code from a developer's point of view. This will support developers in coping with the challenge of implementing a software system and its user interface. We have shown how the approach can be put into practice and how it achieves its goal. DEUCE, which is a proof-of-concept tool suite, is based on a declarative formalism to specify the different user interface concerns at various levels of abstraction. The core of DEUCE contains a backward chainer. This reasoning mechanism combines the specifications and constructs a user interface that co-operates with the underlying application. It uses the VisualWorks Smalltalk UI framework, method-wrappers and a constraint solver to do so. As a target programming language we have chosen VisualWorks Smalltalk. Hence the UI primitives that are used are part of the VisualWorks UI framework. This is a good representative development environment since it is based on the well-known MVC paradigm. The UI development tools contain a representative set of widgets and UI actions that are present in all major development platforms.

In the following we recapitulate our findings and highlight the main contributions, after which we discuss future research directions. We end with enumerating technical considerations for the implementation of DEUCE.

7.1 Contributions

In current software development environments there is a clear lack of support to separate user interface and application. In this dissertation we addressed this problem and we have contributed

- to the terminology of user interface concerns,
- to the conceptual solution towards separation of user interface concerns, and
- with a proof-of-concept implementation to validate the conceptual solution,
- with support for VisualWorks Smalltalk user interfaces.

7.1.1 User Interface Concerns

A first contribution of this dissertation is the careful analysis of terminology related to the separation of user interface concerns (see Chapter 3). We also defined the various UI concerns that exist. Although existing approaches offer a similar concern division, there are significant differences to bear in mind. These mainly result from the fact that current approaches pay little or no attention to the code tangling that results from linking UI and application with each other. The user interface concerns we defined are:

- the presentation logic, which specifies everything that is purely UI related, namely its visualisation and its behaviour.
- the application logic, which specifies what behaviour from the underlying application can be called from within the UI, and what behaviour of the application triggers the UI.
- the connection logic, which specifies how application logic and presentation logic are combined and interact.

As described in Chapter 3, these three concerns should be specified in complete isolation from each other. Not only conceptually but also at the implementation level they should be. Furthermore the underlying application must remain oblivious to these user interface concerns. Obviously the user interface concerns themselves are not oblivious to the underlying application as the interface is to interact with the application. Nevertheless this knowledge is concentrated in the application logic concern.

7.1.2 A Conceptual Solution for Separating UI Concerns

In Chapter 2 we analysed existing work and discussed the main players of the field that focus on achieving a separation. Unfortunately existing approaches fall short to obtain a clean separation when analysed in the context of new software challenges discussed in the introduction of this dissertation. In short, these software systems require linking the application and the UI in both directions, the possibility to use different views for the same application, and user interfaces that are subject to runtime changes. As discussed in Chapter 3, current approaches fail in one or more of these regards.

In this dissertation we elicited a set of requirements for a framework to achieve a separation of concerns, namely:

- **Requirement 1:** *A separate specification for each concern*
- **Requirement 2:** *Specifications at a high-level of abstraction*
- **Requirement 3:** *A mapping from high-level entities onto code-level entities*
- **Requirement 4:** *An automated way to combine the different UI concerns into a complete and final application*
- **Requirement 5:** *Provision for automated layout.*

Requirement 1 forms the cornerstone for the modularisation of concerns as it separates them and allows for reusing or evolving either of the concerns without affecting the other ones. As a lot of code tangling in user interfaces results from the fact that the application and UI interactions are implemented at a low-level, requirement 2 aids developers in abstracting from the implementation details and allows us to focus on the functionality of the concern itself. Consequently these high-level specifications need to be transformed into a low-level specification without again exposing developers to the implementation details, which brings us to requirement 3. Although the mapping is provided by someone at some point, it can be reused for all user interfaces for that particular implementation level (or platform). This turns it into an asset that will ease the future development of a multitude of applications. Additionally requirement 4 ensures that the final application is composed without further developers' intervention. Finally automated layout (requirement 5) is not to be overlooked. This enables developers to change the UI visualisation at runtime without needing to entangle programmatic UI changes in the base application code.

7.1.3 A Proof-of-concept Implementation

We have shown that the various requirements can be fulfilled technically by implementing a proof-of-concept named DEUCE, which supports the separation of user interface concerns for VisualWorks Smalltalk software systems. In Chapter 6 we validated the solution through several archetypical scenarios that illustrate how UI concerns can be evolved and changed independently from one another, and therefore provide the desired separation of concerns. When using DEUCE for non-Smalltalk platforms, it has to be investigated how the necessary mechanism can be provided. However, as SOUL is used as a declarative medium and also has been linked with Java [Bri07], we expect little difficulties when using DEUCE for Java. It would nevertheless require the mapping libraries (that map high-level entities onto actual Smalltalk entities) to be re-implemented. We will now elaborate on the technical aspects of this dissertation.

A Declarative Language to specify UI Concerns

DEUCE uses a declarative language for specifying user interface concerns. As explained in Chapter 4 Section 4.1, it is intuitive for developers to think about the UI in terms of what it represents instead of how it is achieved. Several other existing UI frameworks have added the notion of declarative specifications, such as JGoodies, Adobe's Adam and Eve, and UIML. Note that not all of these approaches offer the power of specifying advanced user interfaces for they lack an inference engine. Since DEUCE uses a combination of a declarative language and a reasoning engine, it has the additional strength that the declarative UI statements can be used in a reasoning process. This process is used by more expressive and powerful rules as it can derive knowledge from these rules, and hence can transform more abstract rules in low-level UI statements.

Meta-programming Mechanisms behind the Scenes

DEUCE uses several meta-programming mechanisms behind the scenes in order to link the UI and the application. First of all, it uses SOUL as a declarative medium to express UI specifications, which was explained in Chapter 4 Section 4.2. The UI specifications are used to construct a Smalltalk UI. Therefore DEUCE provides a set of rules that describe the construction of such a UI and that transform the high-level UI component descriptions into low-level Smalltalk UI components. These rules use SOUL's meta programming facilities. The underlying Smalltalk level is called from within the logical level which enables the creation and manipulation of objects.

Secondly, DEUCE uses method wrappers to link the application to the UI, such that upon a change in the application the logic level is triggered. The declarative UI specifications describe when the application will launch a call back to the UI. DEUCE transforms these specifications into a method wrapper that will intercept the particular message send to the application. Hence, DEUCE is responsible for injecting the application with triggers to the logical level and uses meta programming as a technique to inject the method wrappers that will do so. This is needed to make the application oblivious of the user interface.

Support for Smalltalk UIs

The rules that map the high-level UI specifications onto low-level code entities are in this case rules that map high-level specifications onto low-level Smalltalk UI entities. Although the set of rules that we have provided is not complete in the sense that we do not support all of Smalltalk's UI components, we have provided an initial set to support the creation of VisualWorks Smalltalk UIs, and more particularly the support for changing and updating UI components dynamically.

7.2 Future Research Directions

With DEUCE we have shown how a separation of user interface concerns can be achieved such that each of the concerns can evolve in separation of the others. Such a separation is not always supported by current approaches to the full extent and is becoming of importance in new software challenges which require programmatic intervention. As we described in the introduction of this dissertation, in current software systems variability has come to play an important role. In future research we would like to combine our experiences with other research directions such that an integrated approach will support developers even better, especially when dealing with these new software challenges.

7.2.1 From UI Design to UI Code

In this dissertation we have proposed the technical platform for achieving a modularisation of user interface code. However, this work can only be successful in the long-run if it is fit into the development process. This requires additional research that is quite different from the work we have presented here. In Chapter 3 we introduced the high-level methodology for creating user interfaces with a solution that provides a clean separation of user interface concerns. When putting this methodology into practice, we experienced the need for programmer's guidelines on how to use DEUCE. The pre-conditioned mind that is used to develop a software system in a tangled and scattered way, needs to adapt itself to think about the several concerns in separation. There is a thin line between what is application logic and what is user interface logic and we believe programmer's guidelines will aid developers in making the distinction. Future research will need to conduct experience studies in order to come up with a first set of guidelines. Obviously these guidelines will also evolve over time as developing software systems with separated user interface code becomes more wide-spread. Additionally, valuable lessons can be learned from approaches that offer a different range of user interface concerns, often in the form of models.

Additionally, DEUCE focusses on separating UI concerns at the level of the UI implementation. It does not consider the modelling of user interfaces. Obviously this modelling phase is of importance when creating an application and its user interfaces. Therefore future research directions for DEUCE include incorporating current research on modelling UIs with DEUCE's concern specifications. For example, the several models that are used in model-based UIs (see Chapter 2) can be used to refine the presentation logic concern.

Apart from models for the different UI aspects, also UI design patterns are an interesting aspect when designing a user interface [Laa03, Sin04, Tid05]. These patterns describe recurring solutions in user interfaces. Design patterns also hint at an implementation solution. It would be interesting to use DEUCE's reuse capabilities for turning these pattern descriptions into reusable UI pattern rules which can be configured for any high-level UI specification. For instance, the CRUD (create-retrieve-update-delete) pattern describes how a set of new, edit, save, and delete buttons behave and are for instance enabled and disabled upon clicking one or the other. By offering the pattern and its UI behaviour as a reusable pattern, other buttons can be linked to this behaviour by describing what role they play in the pattern.

7.2.2 Incorporating Context-Oriented Programming

Context-oriented programming [Cos05] is a fairly new research domain that investigates how different contexts can be specified in separation and are combined at runtime to provide for the application behaviour that is valid for a present set of contexts. It is especially in those context-sensitive systems that UIs need to expose a high degree of flexibility. Furthermore this flexibility is not limited to providing yet another view on the same interface. Existing approaches exist for modelling context and for creating a different UI for each of these contexts (statically). Nevertheless research on ambient intelligent systems hints that also UI behaviour will be influenced by contexts. With DEUCE we have touched context influences that require the UI to adapt dynamically, but future work will include the incorporation of context-oriented programming into our ideas and requirements for the separation of UI concerns.

7.2.3 UI Generation and Model-Driven Engineering

Research on context-sensitive UIs has acknowledged the need for accessing model information at runtime because these systems need to adapt to the context and need to reflect dynamic changes in context in their user interfaces when appropriate [Sot05, VdB05, Pat05, Loh06, Dem06, Cle06]. However, most approaches for automatically creating user interfaces use a generative approach and use the high-level models to generate the user interface code. For instance this is the case in Model-Driven Engineering (e.g. model-based UIs). We believe that our approach is in line with MDE. The UI concern specifications in DEUCE provide for runtime 'models' of the UI, although we agree that the more fine-grained models as used in model-based UIs for instance can be used to fine-tune the presentation logic concern. DEUCE combines (transforms) the runtime models into a working application through means of its reasoning mechanism. Furthermore thanks to the declarative nature of our approach, the 'transformations' can reason about their execution rather than having to provide a meticulous one-to-one mapping.

Although originally we considered declarative meta-programming as a means to generate user interfaces [God02], DEUCE does not generate UI code but creates the UI objects dynamically and interacts with them continuously. This allows to access and reflect upon 'model' information at runtime. However, for performance reasons we believe it might be interesting to re-investigate to what extent UIs can be generated and what (limited set of) infrastructure and information is needed at runtime to reflect the dynamic changes in context. Additionally we want to further explore the role DEUCE could play within the MDE community.

7.2.4 UI Specifications as a Domain Specific Language

The declarative user interface specifications we propose in this dissertation can be considered to be a domain specific language language. The high-level rules and facts are dedicated to user interfaces only, and therefore specific for the domain of UIs. Additionally, one can create his own set of rules that apply for their particular kind of interface domain, e.g. by specifying their own type of components and user interface patterns. We have considered UI specifications to be a domain specific language in the past [God04], but in future work we would like to continue on this path and investigate how this idea can be further explored.

7.2.5 Industrial Validation

The work in this dissertation shows how a separation of concerns can be achieved, and we believe that it will aid developers in creating, maintaining and evolving both software systems and user interfaces. In the past industrial partners have already expressed the desire for an approach that achieves a better separation and allows for more powerful UI descriptions that include reasoning as when to apply certain properties and when to apply others. Obviously DEUCE will need more maturity before it can be applied in an industrial case, but taking this road will endorse our findings. The same accounts for exposing several users to the use of DEUCE. Both these tracks will also support future research on the methodology as was discussed at the beginning of this section. Note that it is possible to introduce DEUCE incrementally for instance by re-factoring or creating certain user interfaces while maintaining the original implementation for other user interfaces. As such it can gradually be introduced in an industrial context.

7.2.6 Other Types of UI Code

In this dissertation we focussed on user interface code for traditional business applications. These applications are typically built from a application's point of view, and its corresponding business behaviour. The user interface is built as an interface between that application and the user. These user interfaces typically provide form-based windows with traditional widgets such as buttons, input fields, drop down menus, and text labels.

Other types of applications will require other types of user interfaces. For example in a drawing program, objects are moved around through direct manipulation. Virtual world applications (e.g. Second Life) need yet another UI approach and are often approached from a user interface point of view. Nevertheless these applications also consist of a user interface and an application part, and therefore can possibly benefit from a separation of both parts as well. However, future work is needed to investigate how the solution we offered in this dissertation can contribute when implementing these other kinds of user interface code.

7.3 Future Implementation Improvements for DEUCE

Chapter 5 explained the underlying mechanisms used by DEUCE. These mechanisms were chosen as they provide the necessary infrastructure to support a solution for the separation of user interface concerns. However, in future work other techniques can be explored as they might introduce additional benefits or solve limitations of the current DEUCE implementation.

7.3.1 Minimal Technical Requirements

By implementing DEUCE as a proof-of-concept for the conceptual solution proposed in Chapter 3, we show that it is possible to provide for such a solution. DEUCE uses powerful implementation mechanisms and techniques in order to provide a maximum of expressiveness. As such this work was not limited by technical issues. We acknowledge that some of these technologies are too advanced for the everyday developer and can introduce additional pitfalls. Further research will need to look into the minimal requirements that are necessary to implement the conceptual solution we propose. As such, the minimal technical overhead for both implementing and using DEUCE will be established.

7.3.2 Modularising Logic

In Section 5.9 we explained how logic is currently modularised in SOUL. In short, SOUL uses logic layers as static modules to store facts and rules. In general repositories consist of several layers, the facts and rules of which are loaded in the repository for use during the runtime reasoning process.

DEUCE uses the layers to separate the concerns by dedicating a layer to each of the concerns. Within a concern, several layers can be used to provide for different levels of abstraction and UI specifications. By plugging in and out layers into repositories, the relevant layers for a certain user interface are combined into a whole. Although this layer based modularisation was sufficient for the proof-of-concept DEUCE implementation, we want to look into rule management mechanisms used in other large scale declarative systems, such as expert systems. When introducing large sets of UI specifications (rules and facts), an improved rule-management system will be required in order to select the relevant rule layers and plug in and out the required logic statements.

7.3.3 Backward Versus Forward Chained Reasoning

Declarative programming languages use inferencing as a process for deriving information from known (and assumed) information (i.e. facts and rules). Two important inference techniques are forward chaining (data-driven reasoning) and backward chaining (goal-driven reasoning). In a data-driven reasoning process, the conditions of a rule are matched with the facts in the fact-base. If the conditions are satisfied, the actions are performed. In a goal-driven reasoning process, the possible conclusions of a rule are matched since these are goals achieved by using the rule. The conditions are only tested if the rule can contribute to the achievement of the goal.

Just like Prolog, SOUL uses a backward chained inferencing mechanism. Although this mechanism is sufficient for DEUCE to support the programmer in separating UI concerns, during our experiments we have come to believe that DEUCE could benefit from a forward chained inference mechanism. User interfaces respond to dynamic interactions. Upon such an interaction DEUCE triggers the reasoning process by calling upon a certain query. When using a forward chainer, such an interaction would result in new knowledge being added to the factbase. When new knowledge is added, new conclusions and facts can be inferred. In turn, this new knowledge leads to other UI and application actions being triggered. In future work, we will investigate what the benefits are of opting for a forward chainer (data-driven approach) over a goal-driven approach. Additionally we will look into the effects of a possible loss in power of expressiveness. Currently the use of a data-driven approach is investigated in the context of highly context-sensitive systems such as ambient intelligence systems where new context information becomes valid (and invalid) rapidly and the software system is required to react to these context changes [Mos07].

7.3.4 Traditional Aspect-Oriented Programming Techniques

In Chapter 2 we referred to Aspect-Oriented Programming (AOP) as a technique for achieving a separation of concerns, although it has not been applied to user interface concerns as yet. In AOP, aspects are used to express cross-cutting concerns throughout a base program. Aspects haven been used to gather information about the UI and application dynamics [Li08]. Although this approach does not create a separation of user interface concerns, it does consider the user interface to be an important and cross-cutting concern. Obviously so does this dissertation and although DEUCE does not use a traditional pointcut language or a traditional weaver, it provides an aspect-oriented approach for dealing with user interface concerns. The various concerns are specified in isolation and an underlying mechanism is provided to compose these concerns and the underlying application into the final software system. In future work it is nevertheless interesting to look at these traditional AOP techniques and at how DEUCE can benefit from using them.

In order to link the application with the UI, DEUCE uses method wrappers for instrumenting the application with the necessary code fragments that trigger the declarative reasoning process. Although method wrappers provide sufficient expressiveness for the current purpose of DEUCE, other pointcut languages might provide a stronger mechanism to capture method calls, and hence application events. This extra expressiveness enables the creation of fine grained application events. For example some aspect languages allow specifying that an aspect applies for all methods with a certain name without considering the class they are defined in. Using this we could specify that all methods in an application that add something to their database by sending a message that contains the ‘add’ keyword in its name, should trigger the logic level in order to update the UI.

Note that the method wrappers technique we have adopted, has been used to build aspect-oriented languages, for instance in AspectS [Hir03]. AspectS provides aspect-oriented language features and introduces an abstraction layer on top of method wrappers. These abstractions would already allow DEUCE to specify more fine grained application events. However similar (and other) abstractions can be added to DEUCE as well by providing the necessary abstraction rules.

Some AOP languages provide richer pointcut languages that would allow instrumenting the application code more precisely. This is for instance the case with CARMA's pointcut language which is based on logic programming [Gyb01, Gyb03]. It allows specifying more precisely when to weave and when not to weave an aspect through logic statements. In DEUCE it would be of use to specify conditional application events. For instance, we can choose to trigger the logical level when an application event occurs for a user interface in which certain input fields were entered. If the inputfield condition fails, the application event does not apply, and hence the aspect (that will contain the code to trigger DEUCE) does not need to be woven. Furthermore CARMA weaves aspects dynamically which will allow the link between application code and UI to change dynamically as well. These dynamic link updates are now taken care of in DEUCE through the specification of the actual connection queries that are triggered by the application.

7.3.5 Rule Libraries and Tool Support

DEUCE supports developers in creating user interfaces and maintaining and evolving these interfaces. It provides high-level specifications for the different UI concerns in separation of one another to do so. These high-level specifications are transformed by DEUCE into an actual running application with interface. Obviously DEUCE needs to know how high-level entities map onto low-level entities. These mappings are sets of rules to be used during the reasoning process. Once a mapping has been established for a low-level platform or implementation language, they can be reused for several UIs. Future work on DEUCE consists of extending the current mapping for VisualWorks Smalltalk, as well as other mappings to be created. Note that it is currently possible to call Java from within SOUL [Bri07] which would make it possible to reuse part of the existing DEUCE implementation.

Additionally, adding tool support on top of the declarative statements will allow developers to use dedicated tools for dedicated tasks. For instance developers can create the UI visualisation with a painter like tool, describe the layout relations by drawing connections between components, use state diagrams to specify UI behaviour and conditions that apply when triggering this behaviour, etc. As each of the tools can translate its input into declarative statements, DEUCE will continue to exist behind the scenes to combine the declarative statements into an actual application. The developers however can use the special purpose formalism of their choice for expressing each of the concerns.

7.3.6 Debugging DEUCE

The developers using DEUCE should be supported in dealing with unexpected solutions. For example, when two layout specifications contradict or when the UI is updated dynamically in two different ways, a debugger can aid in fine-tuning the specifications.

Additionally, using a declarative approach for achieving a modularisation of user interface code, requires to deal with incomplete solutions. Currently, if the UI specification leads to more than one possible user interface, DEUCE returns the first solution that is found. Nevertheless, it might be advisable to aid developers in constructing a non-ambiguous UI. On the other hand, if no perfect solution is found, the current implementation of DEUCE will fail. A different approach might be to allow UIs that do not fulfill the specification completely but approximate the specification the most.

7.4 Conclusion

Current software engineering practices lack support for a full separation of user interface concerns. When implementing a software system, developers are confronted with tangled and scattered code. The application and its user interface are strongly interconnected such that evolving a system requires developers to understand both parts thoroughly. Additionally the user interface code is scattered throughout the application code and vice versa such that developers need to scan carefully to find all occurrences of the behaviour they are trying to evolve or maintain.

Furthermore implementing a software system requires developers to understand how the implementation mechanisms work. The need to know how to apply the mechanisms that link the user interface and the application. They also need to know, and remember, the implementation details on how to access the user interface and the user interface components.

As such, current software engineering practices do not support developers in creating user interfaces. They lack a good separation of concerns. In this dissertation we have given a foundation to alleviate the problem. Furthermore we implemented a proof-of-concept for this solution by which we have shown that the conceptual solution can be implemented and does provide for the required separation of concerns. Consequently, the problem of scattered and tangled code is solved and developers are supported in creating, but especially in evolving and maintaining an application and its user interfaces.

Appendix A
DEUCE Rules for Creating the UI

Figure A.1 recapitulates the process of creating a Smalltalk UI based on the high-level UI specifications.

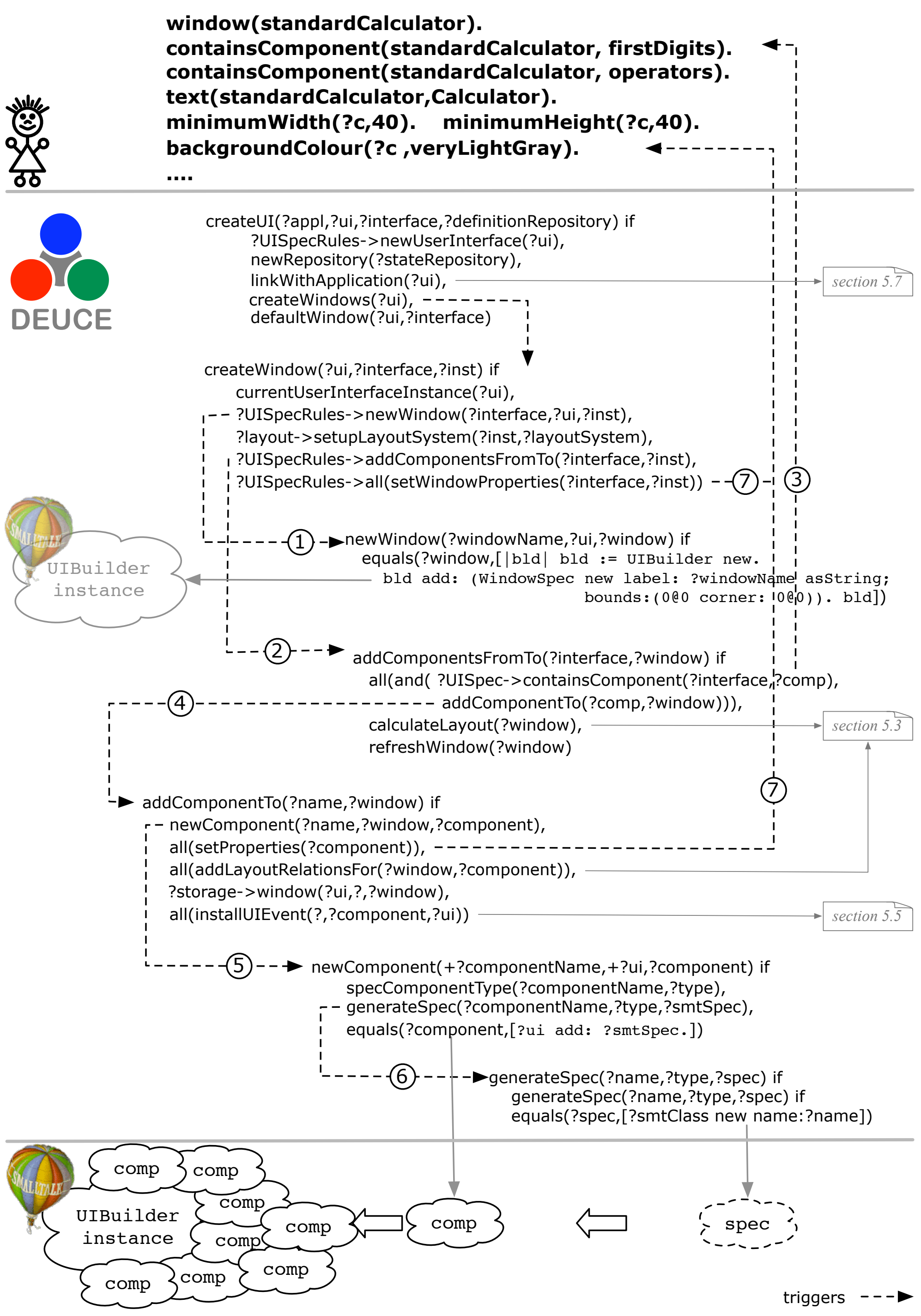
A.1 Accessing the UI Specification

In order to create Smalltalk components based on the high-level UI specification, this specification needs to be accessed in order to retrieve the right information. This is accomplished by the following rules.

```
isComponentInInterface(?comp,?interface) if
        ?UISpec->containsComponent(?interface,?comp)
```

```
specComponentType(?compName,button) if
        ?UISpec->button(?compName).

specComponentType(?compName,inputField) if
        ?UISpec->inputField(?compName).

specComponentType(?compName,label) if
        ?UISpec->label(?compName).

specComponentType(?compName,textEditor) if
        ?UISpec->textEditor(?compName).

specComponentType(?compName,comboBox) if
        ?UISpec->comboBox(?compName).

specComponentType(?compName,dataSet) if
        ?UISpec->dataSet(?compName)
```

```
specFont(?name,?fontName) if
        ?UISpec->font(?name,?fontName),!
```



```
window(standardCalculator).
containsComponent(standardCalculator, firstDigits).
containsComponent(standardCalculator, operators).
text(standardCalculator,Calculator).
minimumWidth(?c,40).    minimumHeight(?c,40).
backgroundColour(?c ,veryLightGray).
....
```

```
createUI(?appl,?ui,?interface,?definitionRepository) if
        ?UISpecRules->newUserInterface(?ui),
        newRepository(?stateRepository),
        linkWithApplication(?ui),
        createWindows(?ui),
        defaultWindow(?ui,?interface)

createWindow(?ui,?interface,?inst) if
      currentUserInterfaceInstance(?ui),
      ?UISpecRules->newWindow(?interface,?ui,?inst),
      ?layout->setupLayoutSystem(?inst,?layoutSystem),
      ?UISpecRules->addComponentsFromTo(?interface,?inst),
      ?UISpecRules->all(setWindowProperties(?interface,?inst))

newWindow(?windowName,?ui,?window) if
      equals(?window,[|bld| bld := UIBuilder new.
          bld add: (WindowSpec new label: ?windowName asString;
                                   bounds:(0@0 corner: 0@0)). bld])

addComponentsFromTo(?interface,?window) if
      all(and( ?UISpec->containsComponent(?interface,?comp),
               addComponentTo(?comp,?window))),
      calculateLayout(?window),
      refreshWindow(?window)

addComponentTo(?name,?window) if
      newComponent(?name,?window,?component),
      all(setProperties(?component)),
      all(addLayoutRelationsFor(?window,?component)),
      ?storage->window(?ui,?,?window),
      all(installUIEvent(?,?component,?ui))

newComponent(+?componentName,+?ui,?component) if
      specComponentType(?componentName,?type),
      generateSpec(?componentName,?type,?smtSpec),
      equals(?component,[?ui add: ?smtSpec.])

generateSpec(?name,?type,?spec) if
      generateSpec(?name,?type,?spec) if
      equals(?spec,[?smtClass new name:?name])
```

Figure A.1: From high-level user interface specification to Smalltalk user interface

```
specText(?compName,?text) if
        ?UISpec->text(?compName,?text)
```

```
specType(?compName,?type) if
        ?UISpec->type(?compName,?type)
```

A.2 Creating Smalltalk UI Components

Based on the high-level component specifications (such as its type and properties), a new Smalltalk UI component is created. To do so a `spec` object is created. Upon adding this spec object to the Smalltalk UI window (a builder instance), Smalltalk transforms it into a component object.

```
newComponent(+?componentName,+?ui,?component) if
   specComponentType(?componentName,?type),
   generateSpec(?componentName,?type,?smtSpec),
   equals(?component,[?ui add: ?smtSpec.])
```

How to generate the Smalltalk spec object depends on the type of the corresponding component.

```
generateSpec(-?name,?type,?spec) if
   smalltalkClassForSpecType(?type,?smtClass),
   equals(?spec,[?smtClass new]),!.

generateSpec(?name,menuButton,?spec) if
   smalltalkClassForSpecType(menuButton,?smtClass),!,
   equals(?spec,[?smtClass new name:?name]).

generateSpec(?name,inputField,?spec) if
   smalltalkClassForSpecType(inputField,?smtClass),!,
   specType(?name,?type),
   equals(?spec,[?smtClass new name:?name;  type:?type; flags: 0]).

generateSpec(?name,comboBox,?spec) if
   smalltalkClassForSpecType(comboBox,?smtClass),!,
   specType(?name,?type),
   equals(?spec,[?smtClass new name:?name;  type:?type; flags: 0]).

generateSpec(?name,textEditor,?spec) if
   smalltalkClassForSpecType(textEditor,?smtClass),
   equals(?spec,[self halt. ?smtClass new name:?name]),!,.

generateSpec(?name,?type,?spec) if
```

```
   smalltalkClassForSpecType(?type,?smtClass),
   equals(?spec,[?smtClass new name:?name]),!.

generateSpec(?name,?abstractType,?spec) if
   ?UISpec->?abstractType(?name,?contents),
   findall(?subspecs,and(includes(?type,?contents),
   generateSpec(?,?type,?subspecs)),?subComponents),
   listAsCollection(?subComponents,?col),
   equals(?spec,
     [CompositeSpecCollection new name: ?name; collection: (?col asArray);
                          compositeSpec: (CompositeSpec new name: ?name)])
```

```
smalltalkClassForSpecType(ActionButtonSpec,[ActionButtonSpec]).
smalltalkClassForSpecType(button,[ActionButtonSpec]).
smalltalkClassForSpecType(inputField,[InputFieldSpec]).
smalltalkClassForSpecType(label,[LabelSpec]).
smalltalkClassForSpecType(textEditor,[TextEditorSpec]).
smalltalkClassForSpecType(menuButton,[MenuButtonSpec]).
smalltalkClassForSpecType(comboBox,[ComboBoxSpec]).

smalltalkClassForSpecType(?type,?class) if
   equals(?class, [Smalltalk at: (?type asString capitalized, 'Spec')
           ifAbsent: [nil]]),
   not(equals(?class,[nil]))
```

The initial component properties are set.

```
setProperties(?component) if
   setText(?component).

setProperties(?component) if
   setFont(?component).

setProperties(?component) if
   setColour(?component)
```

```
setColour(?component) if
   name(?component,?name),
   ?UISpec->foregroundColour(?name,?colour),
   foregroundColour(?component,?colour).

setColour(?component) if
   name(?component,?name),
   ?UISpec->backgroundColour(?name,?colour),
   backgroundColour(?component,?colour).
```

```
setFont(?component) if
   name(?component,?name),
   specFont(?name,?fontName),
   font(?component,?fontName).

setText(?component) if
   name(?component,?name),
   specText(?name,?text),
   text(?component,?text).

setInitialLayout(?component) if
   [?component  component layout: (?component component bounds). true]
```

A.3 Creating a Smalltalk UI Window

The Smalltalk UI (and its main window) is an instance of the UIBuilder class.

```
newWindow(+?windowName,+?ui,?window) if
   equals(?window,[|builder| builder := UIBuilder new.
      builder add: (WindowSpec new label: ?windowName asString;
                                   bounds: (0@0 corner: 0@0)).
      builder userInterface: ?ui.
      builder doFinalHookup.])
```

Components are added and removed from this UIBuilder instance and the window's layout is recalculated.

```
addComponentsFromTo(?interface,?window) if
   all(and(isComponentInInterface(?comp,?interface),
           addComponentTo(?comp,?window))),
   calculateLayout(?window),
   refreshWindow(?window).

removeComponentsFromFrom(?interface,?window) if
   all(and(isComponentInInterface(?comp,?interface),
   removeComponentFrom(?comp,?window))),
   calculateLayout(?window),
   refreshWindow(?window)
```

Upon adding a component to a UI, its initial properties are set, layout relations are added to the window's constraint solver, and the components are linked to the DEUCE specification (`installUIEvent`).

```
addComponentTo(?name,?window) if
   ?UISpec->group(?name,?grcomps),!,
   all(and(includes(?comp,?grcomps),
   addComponentTo(?comp,?window))).

addComponentTo(?name,?window) if
   newComponent(?name,?window,?component),
   all(setInitialLayout(?component)),
   all(setProperties(?component)),
   all(addLayoutRelationsFor(?window,?component)),
   ?storage->window(?ui,?,?window),
   all(installUIEvent(?,?component,?ui))
```

Upon removing a component from a UI, also the layout relations for that component are removed from the window's constraint solver.

```
removeComponentFrom(?name,?window) if
   ?UISpec->group(?name,?grcomps),
   all(and(includes(?comp,?grcomps),
   removeComponentFrom(?comp,?window))).

removeComponentFrom(?component,?window) if
   removeComponent(?component,?window),
   removeLayoutRelationsFor(?window,?component)
```

The initial window properties are set with:

```
setWindowProperties(?windowName,?window) if
   setWindowSize(?windowName,?window).

setWindowProperties(?windowName,?window) if
   setWindowName(?windowName,?window).

setWindowProperties(?windowName,?window) if
   ?UISpec->backgroundColour(?windowName,?color),
   [?window window background: ?color. true].

setWindowName(?windowName,?window) if
   ?UISpec->text(?windowName,?name),
   windowName(?window,?name)

setWindowSize(?name,?window) if
   ?UISpec->windowWidth(?name,?w),
   ?UISpec->windowHeight(?name,?h),!,
   setWindowSize(?window,?w,?h).
setWindowSize(?name,?window) if
   expandedWindowSize(?window)
```

A.4 Steering the Creation Process

The DEUCE process is started by the following query. It stores the definition and state repositories, keeps track of the underlying application and creates the new user interface. The application is linked with the DEUCE logic and the necessary UI windows are created.

```
createUI(?appl,?ui,?interface,?definitionRepository) if
   ?UISpecRules->newUserInterface(?ui),
   newRepository(?stateRepository),
   ?storage->add(userInterface,?ui),
   ?storage->definition(?ui,?definitionRepository),
   ?storage->state(?ui,?stateRepository),
   ?storage->application(?ui,?appl),
   linkWithApplication(?ui),
   createWindows(?ui),
   defaultWindow(?ui,?interface)
```

```
linkWithApplication(?ui) if
   ?UISpecRules->all(installApplicationEvent(?,?ui))
```

```
createWindows(?ui) if
   findall(?a,and(isWindow(?a),createWindow(?ui,?a,?inst)),?as).

createWindow(?ui,?interface,?inst) if
   currentUserInterfaceInstance(?ui),
   ?UISpecRules->newWindow(?interface,?ui,?inst),
   ?layout->setupLayoutSystem(?inst,?layoutSystem),
   ?storage->window(?ui,?interface,?inst),
   ?storage->layoutSystem(?inst,?layoutSystem),
   ?UISpecRules->addComponentsFromTo(?interface,?inst),
   ?UISpecRules->all(setWindowProperties(?interface,?inst)).

defaultWindow(?ui,?interface) if
   [?ui defaultWindow: ?interface. true]
```

Appendix B

DEUCE Rules for Accessing the UI

The runtime Smalltalk UI (and its application) will be accessed from within DEUCE in order for instance to update the UI dynamically or in order to query the UI or application for information. The following predicates are part of the DEUCE core logic and are used to change the properties of components. Note that this is not an exhaustive set of predicates.

B.1 Platform Independent Rules

Changing the property of a component is expressed by referring to the component's name as it is used in the UI specification. Hence, the component with that name is retrieved from the interface and the corresponding property rule is triggered. The latter rule is part of the set of DEUCE logic rules that are specific for the corresponding user interface framework (currently the VisualWorks Smalltalk UI Framework).

```
enable(?compName,?ui) if
   componentWithNameIn(?component,?compName,?ui),
   enable(?component)

disable(?compName,?ui) if
   componentWithNameIn(?component,?compName,?ui),
   disable(?component)
```

```
backgroundColour(?compName,?ui,?colour) if
   componentWithNameIn(?component,?compName,?ui),
   backgroundColour(?component,?colour)

foregroundColour(?compName,?ui,?colour) if
   componentWithNameIn(?component,?compName,?ui),
   foregroundColour(?component,?colour)
```

```
text(?compName,?ui,?text) if
   componentWithNameIn(?component,?compName,?ui),
   text(?component,?text)
```

```
valueForComponent(+?compName,?ui,?value) if
   componentWithNameIn(?comp,?compName,?ui),
   valueForComponent(?comp,?value)
```

B.2 Platform Dependent Rules

Updating Smalltalk component properties is established by sending a Smalltalk message to that component.

B.2.1 Updating Smalltalk Component Properties

```
disable(?component) if
   [?component disable. true].

enable(?component) if
   [?component enable. true]
```

```
name(+?component,?compName) if
   equals(?compName,[?component spec name])
```

```
layout(+?component,+?layout) if
   [?component layout: ?layout. true].

layout(+?component,-?layout) if
   equals(?layout,[?component component layout])
```

```
backgroundColour(+?component,+?colour) if
   smalltalkColourFor(?colour,?value),
      [|prefs| prefs := ?component lookPreferences.
        prefs setBackgroundColor: ?value.
        ?component lookPreferences: prefs. true]


foregroundColour(+?component,+?colour) if
   smalltalkColourFor(?colour,?value),
      [|prefs| prefs := ?component lookPreferences.
         prefs setForegroundColor: ?value.
         ?component  lookPreferences: prefs. true].
```

```
foregroundColour(+?component,-?colour) if
   equals(?colour,
      [ |prefs| prefs := ?component lookPreferences.
        prefs foregroundColor])
```

```
font(?component,?fontName) if
   smalltalkComponentType(?component,ComboBoxView),!.

font(?component,?fontName) if
  [(UILookPolicy new) setStyleOf:(?component widget) to:(?fontName asSymbol).
   ?component widget invalidate. true]
```

```
text(+?component,-?text) if
   equals(?text,[?component widget model value]),!.

text(?component,?text) if
   smalltalkComponentType(?component,inputField),!,
   [ ?component widget editText: (?text asString).
     ?component widget model value: ?text .
     ?component widget invalidate. true].

text(?component,?text) if
   smalltalkComponentType(?component,sequenceView),!.

text(?component,?text) if
   smalltalkComponentType(?component,TextEditorView),!,
   [ ?component widget editText: (?text asString).
     ?component widget model value: ?text.
     ?component widget invalidate. true].

text(?component,?text) if
   smalltalkComponentType(?component,DataSetView),!.

text(?component,?text) if
   smalltalkComponentType(?component,ComboBoxView),!.

text(?component,?text) if
   smalltalkComponentType(?component,WinXPMenuButtonView),!,
   [ ?component widget makeTextLabelFor: (?text asString).
     ?component widget invalidate. true].

text(?component,?text) if
   smalltalkComponentType(?component,TabControlComposite),!.

text(?component,?text) if
         smalltalkComponentType(?component,TreeView),!.
```

```
text(?component,?text) if
        smalltalkComponentType(?component,SubCanvas),!.

text(?component,?text) if
        smalltalkComponentType(?component,Composite),!.

text(?component,?text) if
        [ ?component widget labelString: (?text asString).
          ?component widget invalidate. true]
```

```
valueForComponent(+?component,-?value) if
   equals(?value,[?component widget model value]).

valueForComponent(+?component,+?value) if
   [ ?component widget model value: ?value.
     ?component widget invalidate. true]
```

B.2.2 Updating Smalltalk Window Properties

```
windowSize(+?ui,+?width,+?height) if
   [?ui window minimumSize: (?width @ ?height).true.]
```

```
windowName(+?ui,+?name) if
   [?ui window label: ?name.true.].

windowName(+?ui,-?name) if
   equals(?name,[?ui window label])
```

```
expandedWindowSize(?ui) if
   [xcoord := 0. ycoord := 0.
   ?ui namedComponents associationsDo:
       [:el | |cr| cr := el value component layout corner.
       (cr x > xcoord) ifTrue: [xcoord := cr x].
       (cr y > ycoord) ifTrue: [ycoord := cr y]].
       ?ui window minimumSize:  xcoord @ ycoord. true]
```

Appendix C

DEUCE Rules for Composition

Composing the various concerns and the application into a final software system, is partially done by DEUCE's reasoning mechanism that is triggered at runtime. This is achieved by installing a piece of code to execute (namely to call DEUCE) upon a UI event or application event. DEUCE uses VisualWorks Smalltalk's event handling system for the first, while method wrappers are used for the latter.

C.1 Linking UI Events to DEUCE

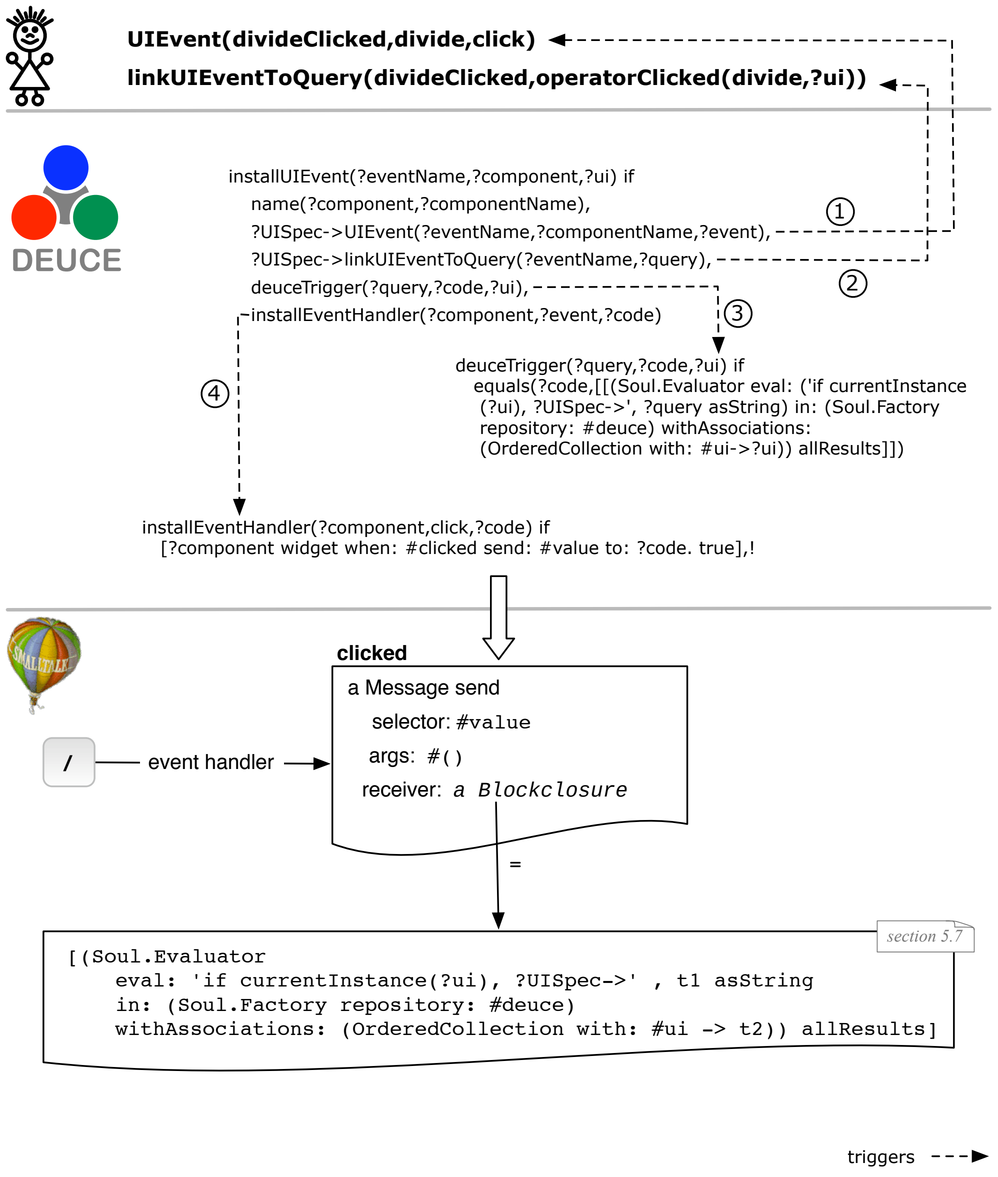
Figure C.1 recapitulates how user interface events are linked with the DEUCE UI specification. The top layer illustrates DEUCE high-level UI specification statements. The DEUCE core logic contains the rules to transform these specifications into the Smalltalk code (blocks) that will trigger the DEUCE reasoning engine.

C.1.1 Platform Independent Logic

```
installUIEvent(?eventName,?component,?ui) if
   name(?component,?componentName),
   ?UISpec->UIEvent(?eventName,?componentName,?event),
   ?UISpec->linkUIEventToQuery(?eventName,?query),
   deuceTrigger(?query,?code,?ui),
   installEventHandler(?component,?event,?code)
```

C.1.2 Platform Dependent Logic

```
installEventHandler(?component,click,?code) if
   [?component widget when: #clicked send: #value to: ?code. true],!.

installEventHandler(?component,getFocus,?code) if
   [?component widget when: #gettingFocus send: #value to: ?code. true],!.

```


```
UIEvent(divideClicked,divide,click)
linkUIEventToQuery(divideClicked,operatorClicked(divide,?ui))
```

```
installUIEvent(?eventName,?component,?ui) if
   name(?component,?componentName),
   ?UISpec->UIEvent(?eventName,?componentName,?event),
   ?UISpec->linkUIEventToQuery(?eventName,?query),
   deuceTrigger(?query,?code,?ui),
   installEventHandler(?component,?event,?code)
```

```
deuceTrigger(?query,?code,?ui) if
  equals(?code,[[(Soul.Evaluator eval: ('if currentInstance
   (?ui), ?UISpec->', ?query asString) in: (Soul.Factory
   repository: #deuce) withAssociations:
   (OrderedCollection with: #ui->?ui)) allResults]])
```

```
installEventHandler(?component,click,?code) if
  [?component widget when: #clicked send: #value to: ?code. true],!
```

```
a Message send
   selector: #value
   args: #()
  receiver: a Blockclosure
```

```
[(Soul.Evaluator
    eval: 'if currentInstance(?ui), ?UISpec->' , t1 asString
    in: (Soul.Factory repository: #deuce)
    withAssociations: (OrderedCollection with: #ui -> t2)) allResults]
```

Figure C.1: From high-level user interface event specification to Smalltalk event handler

```
installEventHandler(?component,?eventName,?code) if
  [?component widget when:(?eventName asSymbol) send: #value to: ?code.true]
```

C.2 Linking Application Events to DEUCE

Figure C.2 recapitulates how application events are linked with the DEUCE UI specification. The top layer shows the high-level specifications that will be transformed by the DEUCE core logic (middle layer) into the Smalltalk specific code instrumentation (through method wrappers).

C.2.1 Platform Independent Logic

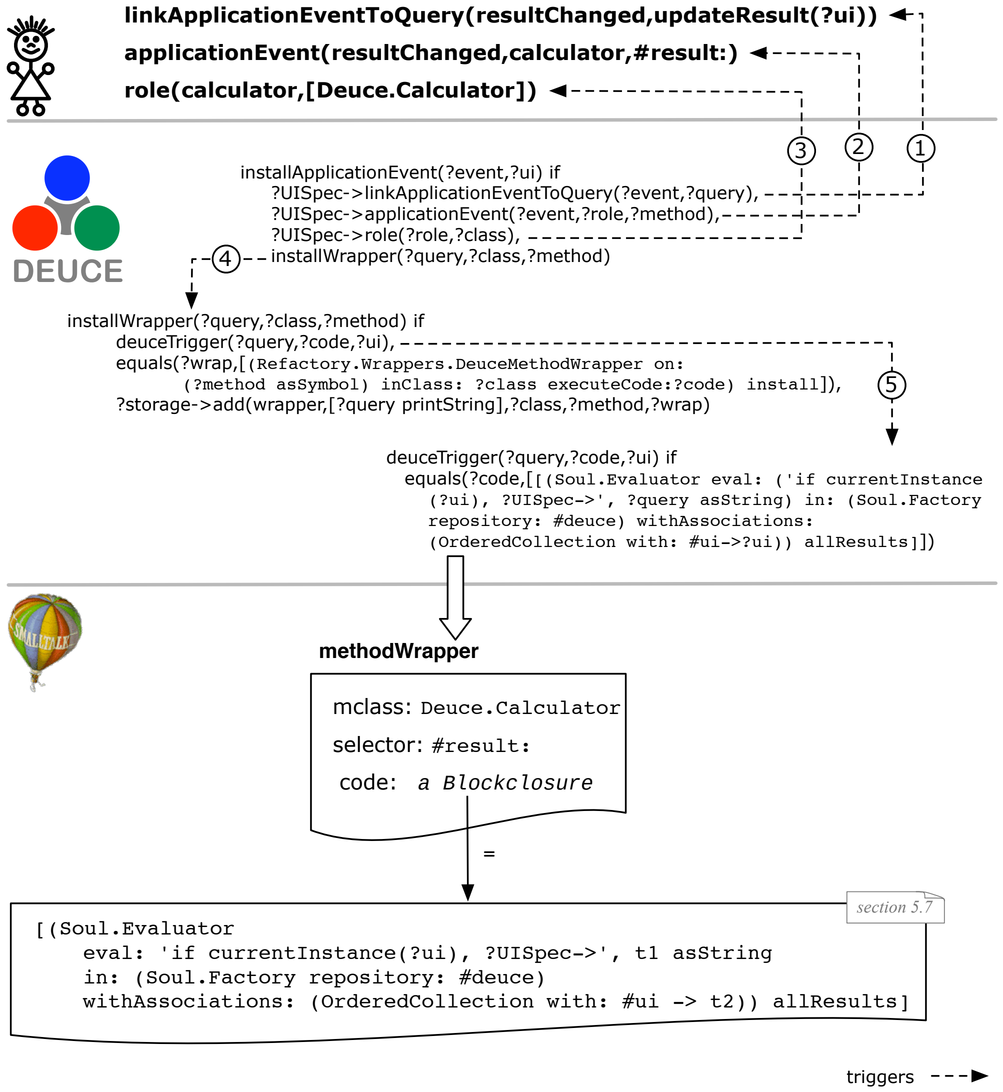
```
installApplicationEvent(?event,?ui) if
   ?UISpec->linkApplicationEventToQuery(?event,?query),
   ?UISpec->applicationEvent(?event,?role,?method),
   ?UISpec->role(?role,?class),
   deuceTrigger(?query,?code,?ui),
   installWrapper(?code,?class,?method)
```

C.2.2 Platform Dependent Logic

```
installWrapper(?code,?class,?method) if
   ?storage->get(wrapper,[?code printString],?class,?method,?wrapper),!.

installWrapper(?code,?class,?method) if
   equals(?wrap, [(Refactory.Wrappers.DeuceMethodWrapper
      on: (?method asSymbol) inClass: ?class executeCode:?code) install.]),
   ?storage->add(wrapper,[?code printString],?class,?method,?wrap)
```

C.3 Triggering DEUCE

```
deuceTrigger(?query,?code,?ui) if
   equals(?code,[[ (Soul.Evaluator
     eval:('if currentUserInterfaceInstance(?ui),?UISpec->',?query asString)
     in:(Soul.Factory repository: #deuce)
     withAssociations:(OrderedCollection with: #ui->?ui)) allResults]])
```



```
linkApplicationEventToQuery(resultChanged,updateResult(?ui))
applicationEvent(resultChanged,calculator,#result:)
role(calculator,[Deuce.Calculator])
```

```
installApplicationEvent(?event,?ui) if
   ?UISpec->linkApplicationEventToQuery(?event,?query),
   ?UISpec->applicationEvent(?event,?role,?method),
   ?UISpec->role(?role,?class),
   installWrapper(?query,?class,?method)
```

```
installWrapper(?query,?class,?method) if
     deuceTrigger(?query,?code,?ui),
     equals(?wrap,[(Refactory.Wrappers.DeuceMethodWrapper on:
            (?method asSymbol) inClass: ?class executeCode:?code) install]),
     ?storage->add(wrapper,[?query printString],?class,?method,?wrap)
```

```
deuceTrigger(?query,?code,?ui) if
  equals(?code,[[(Soul.Evaluator eval: ('if currentInstance
    (?ui), ?UISpec->', ?query asString) in: (Soul.Factory
    repository: #deuce) withAssociations:
    (OrderedCollection with: #ui->?ui)) allResults]])
```

```
mclass: Deuce.Calculator
selector: #result:
code:  a Blockclosure
```

```
[(Soul.Evaluator
    eval: 'if currentInstance(?ui), ?UISpec->', t1 asString
    in: (Soul.Factory repository: #deuce)
    withAssociations: (OrderedCollection with: #ui -> t2)) allResults]
```

Figure C.2: From high-level application event specification to low-level method wrapper

Appendix D
DEUCE Rules for Automated Layout

Figure D.1 recapitulates the process from high-level layout specification to actual Cassowary constraints. In what follows we will list the rules for each of these steps.

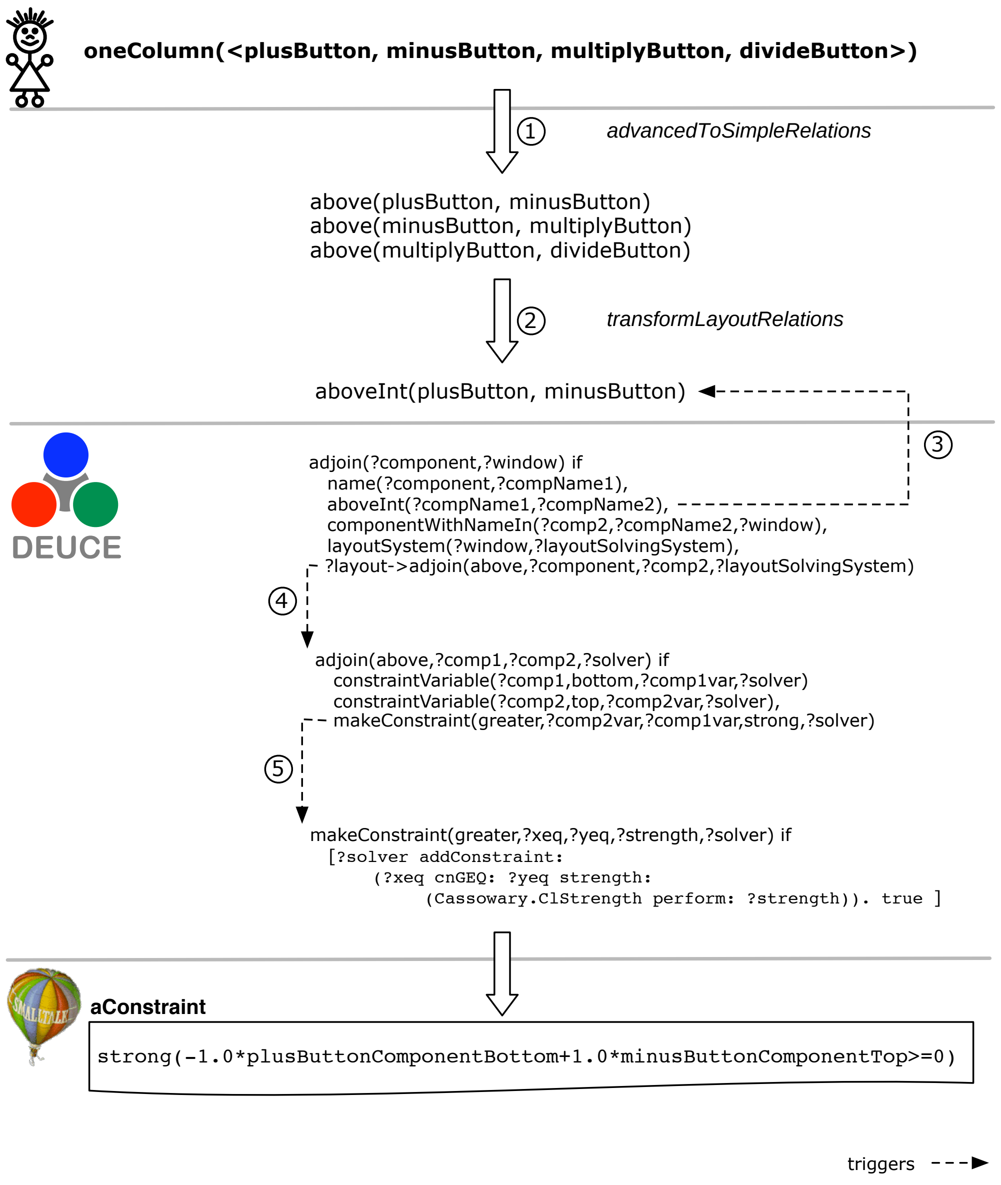
D.1 From Advanced Layout Relations to Basic Layout Relations

DEUCE provides several advanced layout strategies that can be expressed in terms of the basic relations, such as there are one-row, one-column, same-row and same-column.

Components are put into one-row by transforming this specification into a left-of relation between each of the components in that row.

```
oneRow if
   ?UISpec->oneRow(?name),
   ?UISpec->group(?name,?comps),
   oneRowToLeftOf(?comps).

oneRow if
   ?UISpec->oneRow(?comps),
   list(?comps),
   oneRowToLeftOf(?comps).

oneRowToLeftOf(<?firstComponent,?secondComponent>) if
   addSpecification(leftOf(?firstComponent,?secondComponent)).

oneRowToLeftOf(<?firstComponent|?restComponents>) if
   head(?secondComponent,?restComponents),
   addSpecification(leftOf(?firstComponent,?secondComponent)),
   oneRowToLeftOf(?restComponents)
```

The same is achieved for the one-column relation which is expressed in terms of above relations between each of the components in that column.



```
oneColumn(<plusButton, minusButton, multiplyButton, divideButton>)

above(plusButton, minusButton)
above(minusButton, multiplyButton)
above(multiplyButton, divideButton)

aboveInt(plusButton, minusButton)

adjoin(?component,?window) if
   name(?component,?compName1),
   aboveInt(?compName1,?compName2),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?layoutSolvingSystem),
   ?layout->adjoin(above,?component,?comp2,?layoutSolvingSystem)

adjoin(above,?comp1,?comp2,?solver) if
   constraintVariable(?comp1,bottom,?comp1var,?solver)
   constraintVariable(?comp2,top,?comp2var,?solver),
   makeConstraint(greater,?comp2var,?comp1var,strong,?solver)

makeConstraint(greater,?xeq,?yeq,?strength,?solver) if
  [?solver addConstraint:
       (?xeq cnGEQ: ?yeq strength:
            (Cassowary.ClStrength perform: ?strength)). true ]

strong(-1.0*plusButtonComponentBottom+1.0*minusButtonComponentTop>=0)
```

Figure D.1: From high-level layout specification to low-level constraint

```
oneColumn if
   ?UISpec->oneColumn(?name),
   ?UISpec->group(?name,?comps),
   oneColumnToAbove(?comps).

oneColumn if
   ?UISpec->oneColumn(?comps),
   list(?comps),
   oneColumnToAbove(?comps).

oneColumnToAbove(<?firstComponent,?secondComponent>) if
   addSpecification(above(?firstComponent,?secondComponent)).

oneColumnToAbove(<?firstComponent|?restComponents>) if
   head(?secondComponent,?restComponents),
   addSpecification(above(?firstComponent,?secondComponent)),
   oneColumnToAbove(?restComponents)
```

Putting a list of components on the same horizontal line means putting them in the same row and aligning them on the same horizontal line (top).

```
sameRow if
   ?UISpec->sameRow(?name),
   ?UISpec->group(?name,?comps),
   sameRowToLeftOfAndTopAlign(?comps).

sameRow if
   ?UISpec->sameRow(?comps),
   list(?comps),
   sameRowToLeftOfAndTopAlign(?comps).

sameRowToLeftOfAndTopAlign(<?firstComponent,?secondComponent>) if
   addSpecification(leftOf(?firstComponent,?secondComponent)),
   addSpecification(topAlign(?firstComponent,?secondComponent)).

sameRowToLeftOfAndTopAlign(<?firstComponent|?restComponents>) if
   head(?secondComponent,?restComponents),
   addSpecification(leftOf(?firstComponent,?secondComponent)),
   addSpecification(topAlign(?firstComponent,?secondComponent)),
   sameRowToLeftOfAndTopAlign(?restComponents)
```

Putting a list of components on the same vertical line means putting them in the same column and aligning them on the same vertical line (left).

```
sameColumn if
   ?UISpec->sameColumn(?name),
   ?UISpec->group(?name,?comps),
   sameColumnToAboveAndLeftAlign(?comps).

sameColumn if
   ?UISpec->sameColumn(?comps),
   list(?comps),
   sameColumnToAboveAndLeftAlign(?comps).

sameColumnToAboveAndLeftAlign(<?firstComponent,?secondComponent>) if
   addSpecification(above(?firstComponent,?secondComponent)),
   addSpecification(leftAlign(?firstComponent,?secondComponent)).

sameColumnToAboveAndLeftAlign(<?firstComponent|?restComponents>) if
   head(?secondComponent,?restComponents),
   addSpecification(above(?firstComponent,?secondComponent)),
   addSpecification(leftAlign(?firstComponent,?secondComponent)),
   sameColumnToAboveAndLeftAlign(?restComponents)
```

The diagonal relationship can for instance be expressed in terms of the one-column and one-row relationship.

```
diagonal if
   ?UISpec->diagonal(?name),
   diagonalToColumnAndRow(?name).

diagonalToColumnAndRow(?comps) if
   addSpecification(oneRow(?comps)),
   addSpecification(oneColumn(?comps))
```

D.2 Internal Representation for Layout Relations

The basic layout specifications are translated into an internal representation such that all layout relations, also the ones expressed in-between groups, are considered.

```
transformToInternalRelation(?predicate) if
   ?UISpec->?predicate(?group1,?group2),
   groupInt(?group1,?comps1),
   groupInt(?group2,?comps2),
   transformList(?predicate,?comps1,?comps2).

transformToInternalRelation(?predicate) if
   ?UISpec->?predicate(?comp1Name,?comp2Name),
   not(groupInt(?comp1Name,?)),
   not(groupInt(?comp2Name,?)),
   list(?comp1Name),
   list(?comp2Name),
   transformList(?predicate,?comp1Name,?comp2Name).

transformToInternalRelation(?predicate) if
   ?UISpec->?predicate(?comp1Name,?comp2Name),
   not(groupInt(?comp1Name,?)),
   not(groupInt(?comp2Name,?)),
   not(list(?comp1Name)),
   list(?comp2Name),
   transformList(?predicate,<?comp1Name>,?comp2Name).

transformToInternalRelation(?predicate) if
   ?UISpec->?predicate(?group1,?comp2),
   groupInt(?group1,?comps1),
   not(groupInt(?comp2,?)),
   transformList(?predicate,?comps1,<?comp2>).

transformToInternalRelation(?predicate) if
   ?UISpec->?predicate(?comp1,?group2),
   groupInt(?group2,?comps2),
   not(groupInt(?comp1,?)),
   transformList(?predicate,<?comp1>,?comps2).

transformToInternalRelation(?predicate) if
   ?UISpec->?predicate(?comp1Name,?comp2Name),
   not(groupInt(?comp1Name,?)),
   not(groupInt(?comp2Name,?)),
   transformList(?predicate,<?comp1Name>,<?comp2Name>)
```

```
transformList(?predicate,?list1,?list2) if
   findall(?comp1, and(includes(?comp1,?list1),
      transformComponent(?predicate,?comp1,?list2)),?rs)
```

```
transformComponent(?predicate,?comp1,?list2) if
   equals(?pred,[(?predicate asString, 'Int') asSymbol]),
   findall(?comp1,and(includes(?comp2,?list2),
      addSpecification(?pred(?comp1,?comp2))),?result)
```

```
expandGroups if
   expandGroup(?group,?components),
   getRepository(?UISpec,?rep),
   assert(groupInt(?group,?components),?rep)

expandGroup(?group,?components) if
```

```
   ?UISpec->group(?group,?comps),
   findall(?sublist,and(includes(?comp,?comps),
   expandGroup(?comp,?sublist)),?sublists),
   flatten(?sublists,?components).

expandGroup(?group,<?group>) if
   not(?UISpec->group(?group,?comps))
```

D.3 From Basic Layout Relations to Layout System Rules

The first two basic layout relations that can be expressed with DEUCE are above and left-of. Their counterparts above and right-of can be expressed in terms of these two. Each of the rules has several alternatives in order to take into account the spacing-between components, if this was specified in the UI specification. These basic relations are transformed into adjoin relations. The latter are resolved by the automated layout system rules.

```
adjoin(?component,?window) if
   name(?component,?compName1),
   aboveInt(?compName1,?compName2),
   ?UISpec->spacingBetween(?compName1,bottom,?compName2,top,?space),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->adjoin(above,?component,?comp2,?space,?system).

adjoin(?component,?window) if
   name(?component,?compName1),
   aboveInt(?compName1,?compName2),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->adjoin(above,?component,?comp2,?system).

adjoin(?component,?window) if
   name(?component,?compName1),
   aboveInt(?compName2,?compName1),
   ?UISpec->spacingBetween(?compName2,bottom,?compName1,top,?space),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->adjoin(above,?comp2,?component,?space,?system).

adjoin(?component,?ui) if
   name(?component,?compName1),
   aboveInt(?compName2,?compName1),
   componentWithNameIn(?comp2,?compName2,?ui),
   layoutSystem(?ui,?system),
   ?layout->adjoin(above,?comp2,?component,?system)
```

```
adjoin(?component,?window) if
   name(?component,?compName1),
   leftOfInt(?compName1,?compName2),
   ?UISpec->spacingBetween(?compName1,right,?compName2,left,?space),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->adjoin(leftOf,?component,?comp2,?space,?system).

adjoin(?component,?window) if
   name(?component,?compName1),
   leftOfInt(?compName1,?compName2),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->adjoin(leftOf,?component,?comp2,?system).

adjoin(?component,?window) if
   name(?component,?compName1),
   leftOfInt(?compName2,?compName1),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->adjoin(leftOf,?comp2,?component,?system).

adjoin(?component,?ui) if
   name(?component,?compName1),
   leftOfInt(?compName2,?compName1),
   ?UISpec->spacingBetween(?compName2,right,?compName1,left,?space),
   componentWithNameIn(?comp2,?compName2,?ui),
   layoutSystem(?ui,?system),
   ?layout->adjoin(leftOf,?comp2,?component,?space,?system)
```

DEUCE user interface specifications can also make use of the next-to and on-top-of relations in order to place components on the same line horizontally or vertically. While the left-of relation places one component to the left of another, the next-to relation assures the two components are actually also on the same horizontal line.

```
adjoin(?component,?window) if
   name(?component,?compName1),
   nextToInt(?compName1,?compName2),
   ?UISpec->spacingBetween(?compName1,right,?compName2,left,?space),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->adjoin(nextTo,?component,?comp2,?space,?system).
```

```
adjoin(?component,?ui) if
   name(?component,?compName1),
   nextToInt(?compName2,?compName1),
   ?UISpec->spacingBetween(?compName2,right,?compName1,left,?space),
   componentWithNameIn(?comp2,?compName2,?ui),
   layoutSystem(?ui,?system),
   ?layout->adjoin(nextTo,?comp2,?component,?space,?system).

adjoin(?component,?window) if
   name(?component,?compName1),
   onTopOfInt(?compName1,?compName2),
   ?UISpec->spacingBetween(?compName1,bottom,?compName2,top,?space),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->adjoin(onTopOf,?component,?comp2,?space,?system).

adjoin(?component,?ui) if
   name(?component,?compName1),
   onTopOfInt(?compName2,?compName1),
   ?UISpec->spacingBetween(?compName2,bottom,?compName1,top,?space),
   componentWithNameIn(?comp2,?compName2,?ui),
   layoutSystem(?ui,?system),
   ?layout->adjoin(onTopOf,?comp2,?component,?space,?system)
```

Four additional basic layout relations are used to align components.

```
align(?component,?window) if
   name(?component,?compName1),
   bottomAlignInt(?compName1,?compName2),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->align(bottom,?component,?comp2,?system).

align(?component,?window) if
   name(?component,?compName1),
   bottomAlignInt(?compName2,?compName1),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->align(bottom,?comp2,?component,?system).

align(?component,?window) if
   name(?component,?compName1),
   leftAlignInt(?compName1,?compName2),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->align(left,?component,?comp2,?system).
```

```
align(?component,?window) if
   name(?component,?compName1),
   leftAlignInt(?compName2,?compName1),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->align(left,?comp2,?component,?system).

align(?component,?window) if
   name(?component,?compName1),
   rightAlignInt(?compName1,?compName2),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->align(right,?component,?comp2,?system).

align(?component,?window) if
   name(?component,?compName1),
   rightAlignInt(?compName2,?compName1),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->align(right,?comp2,?component,?system).

align(?component,?window) if
   name(?component,?compName1),
   topAlignInt(?compName1,?compName2),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->align(top,?component,?comp2,?system).

align(?component,?window) if
   name(?component,?compName1),
   topAlignInt(?compName2,?compName1),
   componentWithNameIn(?comp2,?compName2,?window),
   layoutSystem(?window,?system),
   ?layout->align(top,?comp2,?component,?system)
```

Other layout specifications are related to the size of components. The UI specifications can specify either or both the minimum and fixed width and height. If both are specified, we have opted to give the largest size the priority, as expressed in the core rules to deal with width and height.

```
width(?component,?ui) if
   name(?component,?compName1),
   ?UISpec->minimumWidth(?compName1,?min),
   ?UISpec->fixedWidth(?compName1,?fix),[?fix > ?min],!,
   layoutSystem(?ui,?system),
   ?layout->size(width,minimum,?component,?min,?system),
   ?layout->size(width,fixed,?component,?fix,?system).
```

```
width(?component,?ui) if
   name(?component,?compName1),
   ?UISpec->fixedWidth(?compName1,?fix),!,
   layoutSystem(?ui,?system),
   ?layout->size(width,fixed,?component,?fix,?system).

width(?component,?ui) if
   name(?component,?compName1),
   ?UISpec->minimumWidth(?compName1,?min),
   layoutSystem(?ui,?system),
   ?layout->size(width,minimum,?component,?min,?system)
```

```
height(?component,?ui) if
   name(?component,?compName1),
   ?UISpec->minimumHeight(?compName1,?min),
   ?UISpec->fixedHeight(?compName1,?fix),[?fix > ?min],!,
   layoutSystem(?ui,?system),
   ?layout->size(height,minimum,?component,?min,?system),
   ?layout->size(height,fixed,?component,?fix,?system).

height(?component,?ui) if
   name(?component,?compName1),
   ?UISpec->fixedHeight(?compName1,?fix),!,
   layoutSystem(?ui,?system),
   ?layout->size(height,fixed,?component,?fix,?system).

height(?component,?ui) if
   name(?component,?compName1),
   ?UISpec->minimumHeight(?compName1,?min),
   layoutSystem(?ui,?system),
   ?layout->size(height,minimum,?component,?min,?system)
```

D.4 From Layout System Rules to Constraint Relations

The layout specifications are transformed into actual constraints for the Cassowary constraint solver.

D.4.1 Positioning relations

The above, left-of, next-to and on-top-of relations are adjoin relations and transformed into constraints. So is the alignment relation. These relations hold between the coordinates of two components.

```
adjoin(above,?x,?y,?solver) if
   constraintVariable(?x,bottom,?xvar,?solver),
   constraintVariable(?y,top,?yvar,?solver),
   makeConstraint(greater,?yvar,?xvar,strong,?solver).

adjoin(above,?x,?y,?space,?solver) if
   constraintVariable(?x,bottom,?xvar,?solver),
   makeEquation(sum,?xvar,?space,?xeq),
   constraintVariable(?y,top,?yvar,?solver),
   makeConstraint(greater,?yvar,?xeq,strong,?solver).


adjoin(leftOf,?x,?y,?solver) if
   constraintVariable(?x,right,?xvar,?solver),
   constraintVariable(?y,left,?yvar,?solver),
   makeConstraint(greater,?yvar,?xvar,strong,?solver).


adjoin(leftOf,?x,?y,?space,?solver) if
   constraintVariable(?x,right,?xvar,?solver),
   constraintVariable(?y,left,?yvar,?solver),
   makeEquation(sum,?xvar,?space,?xeq),
   makeConstraint(greater,?yvar,?xeq,strong,?solver).


adjoin(nextTo,?x,?y,?space,?solver) if
   constraintVariable(?x,right,?xvar,?solver),
   constraintVariable(?y,left,?yvar,?solver),
   makeEquation(sum,?xvar,?space,?xeq),
   makeConstraint(equals,?yvar,?xeq,required,?solver).

adjoin(onTopOf,?x,?y,?space,?solver) if
   constraintVariable(?x,bottom,?xvar,?solver),
   makeEquation(sum,?xvar,?space,?xeq),
   constraintVariable(?y,top,?yvar,?solver),
   makeConstraint(equals,?yvar,?xeq,required,?solver)
```

```
align(?position,?x,?y,?solver) if
   constraintVariable(?x,?position,?xvar,?solver),
   constraintVariable(?y,?position,?yvar,?solver),
   makeConstraint(equals,?yvar,?xvar,required,?solver)
```

D.4.2 Size relations

Sizing relations hold between the co-ordinates of one component.

```
size(height,minimum,?x,?min,?solver) if
   constraintVariable(?x,top,?xvar,?solver),
   constraintVariable(?x,bottom,?yvar,?solver),
   makeEquation(sum,?xvar,?min,?xeq),
   makeConstraint(greater,?yvar,?xeq,strong,?solver).

size(width,minimum,?x,?min,?solver) if
   constraintVariable(?x,left,?xvar,?solver),
   constraintVariable(?x,right,?yvar,?solver),
   makeEquation(sum,?xvar,?min,?xeq),
   makeConstraint(greater,?yvar,?xeq,strong,?solver).

size(height,fixed,?x,?h,?solver) if
   constraintVariable(?x,top,?xvar,?solver),
   constraintVariable(?x,bottom,?yvar,?solver),
   makeEquation(sum,?xvar,?h,?xeq),
   makeConstraint(equals,?yvar,?xeq,required,?solver).

size(width,fixed,?x,?w,?solver) if
   constraintVariable(?x,left,?xvar,?solver),
   constraintVariable(?x,right,?yvar,?solver),
   makeEquation(sum,?xvar,?w,?xeq),
   makeConstraint(equals,?yvar,?xeq,required,?solver)
```

D.4.3 Constraint Variables

Layout relations are expressed between the various boundaries of components. Each of the component boundaries is designated by a constraint variable, generated with the following rules.

```
constraintVariableName(+?comp,+?position,?name) if
   name(?comp,?compName),
   equals(?name,[?compName, ‘-’, ?position]).

possibleConstraintVariablesFor(?compName,?vars) if
   equals(?vars,<[?compName, ‘-’, #top],
      [?compName, ‘-’, #bottom],[?compName, ‘-’, #left],
      [?compName, ‘-’, #right],[?compName, ‘-’, #center]>)
```

D.4.4 Adding Deviation Values

Deviation values are used to express distance between component positions (i.e. constraint variables).

```
makeEquation(sum,?varname,?value,&equation) if
        equals(&equation,[?varname + ?value]).

makeEquation(difference,?varname,?value,&equation) if
        equals(&equation,[?varname - ?value])
```

D.5 From Constraint Relations to Cassowary Constraints

The following rules transform high-level constraint relations into Cassowary constraints. This is done through the use of SOUL's symbiosis with Smalltalk, as the Cassowary constraint solver used is implemented in Smalltalk.

D.5.1 Creating Constraints

```
makeConstraint(greater,?xeq,?yeq,?strength,?solver) if
   [?solver addConstraint: (?xeq cnGEQ: ?yeq strength:
      ((Cassowary.ClStrength perform: ?strength)). true].

makeConstraint(smaller,?xx1,?yx1,?strength,?solver) if
   [?solver addConstraint: (?yx1 cnGEQ: ?xx1 strength:
       (Cassowary.ClStrength perform: ?strength)). true].

makeConstraint(equals,?xeq,?yeq,?strength,?solver) if
   [?solver addConstraint: (?xeq cnEqual: ?yeq strength:
       (Cassowary.ClStrength perform: ?strength)). true]



makeConstraint(stay,?xeq,?strength,?solver) if
   [?solver stay: ?xeq strength:
      (Cassowary.ClStrength perform: ?strength). true]
```

D.5.2 Constraint Solver

The following predicates create a new instance of the Cassowary constraint solver, trigger the system to be solved, and remove the constraints related to a certain component (i.e. variableNames).

```
makeConstraintSolver(?ui,?solver) if
   equals(?solver,[Deuce.ConstraintSolving.ConstraintSolver new]).

solveConstraintSystem(?solver) if
   [?solver solve. true].

removeConstraintsFor(?componentName,?solver) if
   possibleConstraintVariablesFor(?componentName,?v),
   includes(?name,?v),
   [?solver removeConstraintsFor: ?name. true]
```

D.5.3 Constraint Variables

Cassowary constraint variables are either retrieved or created by the following predicates.

```
constraintVariable(?comp,?position,?var,?solver) if
   getConstraintVariable(?comp,?position,?var,?solver),!

getConstraintVariable(?comp,?position,?var,?solv) if
   constraintVariableName(?comp,?position,?name),
   [?solv hasConstraintVariable: ?name],
   equals(?var,[?solv constraintVariable: ?name]).

getConstraintVariable(?comp,?position,?var,?solv) if
   constraintVariableName(?comp,?position,?name),
   [(?solv hasConstraintVariable: ?name)not],
   layout(?comp,?layout),
   equals(?var,[Deuce.ConstraintSolving.LayoutConstraintVariable
      new name: ?name; component: ?comp;
      layout: ?layout ; position: ?position.]),
   [?solv addConstraintVariable: ?var name: ?name. true]
```

List of Figures

List of Tables

Bibliography

[Abr99] M. Abrams, C. Phanouriou, A. Batongbacal, S. Williams, J. Shuster. *UIML: An Appliance-independent XML User Interface Language.* Tech. rep., Harmonia, Inc, 1999.

[Abr04] M. Abrams, J. Heims. *User Interface Markup Language (UIML)*, 2004. Working Draft 3.1.

[App06] Apple Inc. *Cocoa Fundamentals Guide.* Website, 2006. `http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/index.html`.

[App07] Apple Inc. *Cocoa Bindings Programming Topics.* Website, 2007. `http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaBindings/index.html`.

[Bad01] G. J. Badros, A. Borning, P. J. Stuckey. *The Cassowary linear arithmetic constraint solving algorithm.* In ACM Transactions on Computer-Human Interaction, vol. 8(4):pp. 267–306, 2001.

[Bas92] L. Bass, R. Faneuf, R. Little, N. Mayer, B. Pellegrino, S. Reed, R. Seacord, S. Sheppard, M. R. Szczur. *A metamodel for the runtime architecture of an interactive system: the UIMS tool developers workshop.* In SIGCHI Bulletin, vol. 24(1):pp. 32–37, 1992.

[Bot06] G. Botterweck. *A Model-Driven Approach to the Engineering of Multiple User Interfaces.* In A. Pleuss, J. Van den Bergh, H. Hussmann, S. Sauer, A. Boedcher (eds.), *Proceedings of the MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces.* 2006.

[Bow00] A. Bower, B. McGlashan. *Twisting The Triad: The evolution of the Dolphin Smalltalk MVP application framework.* In *Tutorial Paper for ESUG 2000.* 2000.

[Bra98] J. Brant, B. Foote, R. E. Johnson, D. Roberts. *Wrappers to the Rescue.* In E. Jul (ed.), *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, vol. 1445 of *Lecture Notes in Computer Science*, pp. 396–417. Springer, 1998.

[Bre03] R. Breedenraedt. *Step by Step: Event handling in VB.NET.* Website, 2003. `http://www.codeproject.com/vb/net/StepByStepEventsInVBNET.asp`.

[Bri07] J. Brichau, C. De Roover, K. Mens. *Open Unification for Program Query Languages.* In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007).* 2007.

[Car93] L. M. F. Carneiro, M. H. Coffin, D. D. Cowan, C. J. P. Lucena. *User Interface High-Order Architectural Models.* Tech. Rep. 93-14, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, 1993.

[Cle06] T. Clerckx, C. Vandervelpen, K. Luyten, K. Coninx. *A Prototype-Driven Development Process for Context-Aware User Interfaces.* In Lecture Notes in Computer Science LNCS series, vol. 4385:pp. 207 – 214, 2006.

[cod05] *CoDAMoS: Context-Driven Adaptation of Mobile Services.* Website, 2005. `http://www.cs.kuleuven.be/~distrinet/projects/CoDAMoS/`.

[Cos05] P. Costanza, R. Hirschfeld. *Language constructs for context-oriented programming: an overview of ContextL.* In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pp. 1–10. ACM, New York, NY, USA, 2005.

[Cou93] J. Coutaz. *Software architecture modeling for user interfaces.* In *The Encyclopedia of software Engineering.* Wiley and sons, 1993.

[Cou97] J. Coutaz. *PAC-ing the Architecture of Your User Interface.* Tech. rep., CLIPS-IMAG, 1997.

[De 02] W. De Meuter, J. Brichau, K. Mens. *Soul Manual*, 2002. Draft.

[Dea07] N. Deakin. *XUL Tutorial.* Website, 2007. `http://www.xulplanet.com/tutorials/xultu/`.

[Dem06] A. Demeure, G. Calvary, J. Coutaz, J. Vanderdonckt. *The Comets Inspector: Towards Run Time Plasticity Control Based on a Semantic Network.* In K. Coninx, K. Luyten, K. A. Schneider (eds.), *TAMODIA*, vol. 4385 of *Lecture Notes in Computer Science*, pp. 324–338. Springer, 2006.

[Der06] D. Deridder. *A Concept-Centric Environment for Software Evolution in an Agile Context.* Ph.D. thesis, Vrije Universiteit Brussel, 2006.

[Des07] B. Desmet, J. Vallejos, P. Costanza, R. Hirschfeld. *Layered Design approach for Context-Aware Systems.* In *First International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* Limerick, Ireland, 2007. Lero Technical Report 2007-01.

[Dev] DeveloperFusion. *Building Windows Applications - Introduction.* Website. `http://www.developerfusion.co.uk/show/4375/`.

[Dot] DotnetSpider. *What is this .NET all about?* Website. `http://www.dotnetspider.com/tutorials/whatisNet.aspx`.

[Duc01] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, J.-C. Burgelman. *Scenarios for Ambient Intelligence in 2010.* Tech. rep., EC Information Society Technologies Advisory Group (ISTAG), 2001.

[Eve99] M. Evers. *A Case Study on Adaptability Problems of the Separation of User Interface and Application Semantics*, 1999.

[Fil00] R. Filman, D. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness.* In *Workshop on Advanced Separation of Concerns, OOPSLA 2000.* Minneapolis, 2000.

[Fow] A. Fowler. *A Swing Architecture Overview.* Website. `http://java.sun.com/products/jfc/tsc/articles/architecture/`.

[Fow06] M. Fowler. *GUI Architectures.* Website, 2006. `http://www.martinfowler.com/eaaDev/uiArchs.html`.

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns : elements of reusable object-oriented software.* Addison-Wesley, 1995.

[God02] S. Goderis, W. De Meuter. *Generating User Interfaces by means of Declarative Meta Programming.* In *Workshop on Generative Programming, Ecoop 2002.* 2002.

[God04] S. Goderis, D. Deridder. *A Declarative DSL Approach for UI Specification Making UIs Programming Language Independent.* In *Workshop on Evolution and Reuse of Language Specifications for DSLs, Ecoop 2004.* 2004.

[God05a] S. Goderis. *High-level Declarative User Interfaces.* In *OOPSLA Companion Proceedings.* 2005. Presented at the Doctoral Symposium.

[God05b] S. Goderis, D. Deridder. *Declarative User Interfaces : specifying UI variabilities.* In *2nd Workshop on Managing Variabilities Consistently in Design and Code.* 2005.

[God07a] S. Goderis, D. Deridder, E. Van Paesschen. *DEUCE : Separating Concerns in User Interfaces.* In *Proceedings of the Second International Conference on Software Engineering Advances (ICSEA 2007).* 2007.

[God07b] S. Goderis, D. Deridder, E. Van Paesschen, T. D'Hondt. *DEUCE : A Declarative Framework for Extricating User Interface Concerns.* In *Journal of Object Technology*, vol. 6. TOOLS 2007, ETH Zurich, 2007.

[Gul] S. Gul, T. Pavek, R. Kusterer. *Adding Functionality to Buttons: A Beginners Guide.* Website. `http://www.netbeans.org/kb/trails/matisse.html`.

[Gyb01] K. Gybels. *Aspect-Oriented Programming using a Logic Meta Programming Language to express cross-cutting through a dynamic joinpoint structure.* Master thesis, Programming Technology Lab, Vrije Universiteit Brussel, 2001.

[Gyb03] K. Gybels, J. Brichau. *Arranging language features for more robust pattern-based crosscuts.* In *AOSD*, pp. 60–69. 2003.

[Han02] J. Hannemann, G. Kiczales. *Design pattern implementation in Java and AspectJ.* In *2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002,*, vol. 37 of *SIGPLAN Notices*. Seattle, Washington, USA, 2002.

[HI00] J. A. Highsmith III. *Adaptive software development: a collaborative approach to managing complex systems.* Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[Hir03] R. Hirschfeld. *AspectS - Aspect-Oriented Programming with Squeak.* In M. Aksi, M. Mezini, R. Unland (eds.), *Proceedings of Objects, Components, Architectures, Services, and Applications for a Networked World*, vol. LNCS 2591, pp. 216 – 232. Springer-Verlag Heidelberg, 2003.

[Hir08] R. Hirschfeld, P. Costanza, O. Nierstrasz. *Context-oriented Programming.* In Journal of Object Technology (JOT), vol. 7(3):pp. 125–151, 2008.

[Hos01] H. Hosobe. *A modular geometric constraint solver for user interface applications.* In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pp. 91–100. ACM Press, New York, NY, USA, 2001.

[Hun97] J. Hunt. *Smalltalk and Object Orientation : an Introduction.* Springer-Verlag, 1997.

[Insa] Instantiations, Inc. *Data Binding.* Website. `http://download.instantiations.com/DesignerDoc/integration/latest/docs/html/databinding/index.html`.

[Insb] Instantiations, Inc. *Handling Events with the Designer.* Website. `http://download.instantiations.com/DesignerDoc/integration/latest/docs/html/tutorial/handling_events.htm`.

[Insc] Instantiations, Inc. *Instantiations WindowBuilderPro.* Website. `http://www.instantiations.com/windowbuilder/`.

[IST03] IST Advisory Group. *Ambient Intelligence: from vision to reality.* Tech. rep., 2003.

[jGn] *jGnash Personal Finance.* Website. `http://jgnash.sourceforge.net/`.

[jgo] *JGOODIES: Java User Interface Design.* Website. `http://www.jgoodies.com/`.

[Kaz93] R. Kazman, L. Bass, G. Abowd, M. Webb. *Analyzing the Properties of User Interface Software.* Tech. Rep. CS-93-201, Carnegie Mellon University, School of Computer Science, 1993.

[Kel06] A. Kellens, K. Mens, J. Brichau, K. Gybels. *Managing the Evolution of Aspect-Oriented Software with Model-based pointcuts.* In *European Conference on Object-Oriented Programming (ECOOP)*, no. 4067 in LNCS, pp. 501–525. 2006.

[Kop04] C. Koppen, M. Stoerzer. *PCDiff: Attacking the Fragile Pointcut Problem.* In *European Interactive Workshop on Aspects in Software (EIWAS)*. 2004.

[Kow79] R. Kowalski. *Algorithm = logic + control.* In Commun. ACM, vol. 22(7):pp. 424–436, 1979.

[Kra88] G. E. Krasner, S. T. Pope. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* In JOOP, 1988.

[Laa03] S. A. Laakso. *Collection of User Interface Design Patterns.* Website, 2003. `http://www.cs.helsinki.fi/u/salaakso/patterns/`.

[Len04] K. Lentzsch. *The JGoodies Forms Framework.* Tech. rep., 2004.

[Li08] P. Li, E. Wohlstadter. *View-Based Maintenance of Graphical User Interfaces.* to be published in Proceedings of the International Conference on Aspect-Oriented Software Development, 2008.

[Loh06] S. Lohmann, J. W. Kaltz, J. Ziegler. *Dynamic Generation of Context-Adaptive Web Interfaces trhough Model Interpretation.* In A. Pleuss, J. Van den Bergh, H. Hussmann, S. Sauer, A. Boedcher (eds.), *Proceedings of the MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces.* 2006.

[Lok01] S. Lok, S. Feiner. *A Survey of Automated Layout Techniques for Information Presentations.* In *SmartGraphics Symposium*, pp. 61–68. 2001.

[Lu07] X. Lu, J. Wan. *Model Driven Development of Complex User Interface.* In *Third International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2007).* 2007.

[Mei00] E. Meijer, D. van Velzen. *Haskell Server Pages : Functional Programming and the Battle for the Middle Tier.* In *Haskell Workshop 2000.* 2000.

[Mic08] B. Michotte, J. Vanderdonckt. *GrafiXML, A Multi-Target User Interface Builder based on UsiXML.* In *The Fourth International Conference on Autonomic and Autonomous Systems (ICAS 2008).* 2008. IEEE Computer Society Press, March 16-21, 2008 - Gosier, Guadeloupe.

[Mil04] R. Miles. *AspectJ Cookbook.* O'Reilly, 2004.

[Mos07] S. Mostinckx, C. Scholliers, E. Philips, C. Herzeel, W. De Meuter. *Fact Spaces: Coordination in the Face of Disconnection.* In A. L. Murphy, J. Vitek (eds.), *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION)*, vol. 4467 of *Lecture Notes in Computer Science*, pp. 268–285. Springer-Verlag, 2007.

[Moz07a] Mozdev Community Organization. *Creating Applications with Mozilla.* Website, 2007. `http://books.mozdev.org/`.

[Moz07b] Mozilla Developer Center. *Introduction to XUL.* Website, 2007. `http://developer.mozilla.org/en/docs/XUL`.

[MSDa] MSDN Microsoft. *Handling and Raising Events.* Website. `http://msdn2.microsoft.com/en-us/library/edzehd2t(VS.71).aspx`.

[MSDb] MSDN Microsoft. *User Interface Process Application Block for .NET.* Website. `http://msdn2.microsoft.com/en-us/library/ms998252.aspx`.

[MSDc] MSDN Microsoft. *User Interface Process (UIP) Application Block.* Website. `http://msdn2.microsoft.com/en-us/library/ms979213.aspx`.

[Mye00] B. A. Myers, S. E. Hudson, R. F. Pausch. *Past, present, and future of user interface software tools.* In ACM Trans. Comput.-Hum. Interact., vol. 7(1):pp. 3–28, 2000.

[Nig91] L. Nigay, J. Coutaz. *Building user interfaces: organizing software agents.* In ACM (ed.), *Esprit'91 Conference Proceedings.* 1991.

[Par72] D. L. Parnas. *On the criteria to be used in decomposing systems into modules.* In Commun. ACM, vol. 15(12):pp. 1053–1058, 1972.

[Par07] S. Parent, M. Marcus, F. Brereton. *Adobe Source Libraries.* Website, 2007. `http://opensource.adobe.com/`.

[Pat05] F. Paternò. *Model-based tools for pervasive usability.* In Interacting with Computers, vol. 17(3):pp. 291–315, 2005.

[Paw05] R. Pawlak, L. Seinturier, J.-P. Retaillé. *Foundations of AOP for J2EE Development.* Apress, 2005.

[Pin00a] P. Pinheiro da Silva. *User Interface Declarative Models and Development Environments: A Survey.* In P. Palanque, F. Paternò (eds.), *Proceedings of DSV-IS2000*, vol. 1946 of *LNCS*, pp. 207–226. Springer-Verlag, Limerick, Ireland, 2000.

[Pin00b] P. Pinheiro da Silva, T. Griffiths, N. W. Paton. *Generating User Interface Code in a Model Based User Interface Development Environment.* In *Advanced Visual Interfaces*, pp. 155–160. 2000.

[PM07] J. Pérez-Medina, S. Dupuy-Chessa, A. Front. *A Survey of Model Driven Engineering Tools for User Interface Design.* In *Task Models and Diagrams for User Interface Design.* Springer Berlin / Heidelberg, 2007.

[Pot96] M. Potel. *MVP: Model-View-Presenter - The Taligent Programming Model for C++ and Java.* Tech. rep., Taligent Inc, 1996.

[Ree79a] T. Reenskaug. *Models-Views-Controllers.* Tech. rep., Xerox PARC, 1979.

[Ree79b] T. Reenskaug. *Thing-Model-View-Editor : an example from a planningsystem.* Tech. rep., Xerox PARC, 1979.

[Rho06] T. Rho, G. Kniesel. *Independent Evolution of Design Patterns and Application Logic with Generic Aspects - A Case Study, Technical Report IAI-TR-2006-4, Computer Science Department III, University of Bonn.* In *Technical Report IAI-TR-2006-4, Computer Science Department III, University of Bonn.* 2006.

[Sam04] K. Samaan, F. Tarpin-Bernard. *Task models and interaction models in a multiple user interfaces generation process.* In *TAMODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams*, pp. 137–144. ACM, New York, NY, USA, 2004.

[Sch96] E. Schlungbaum. *Model-based User Interface Software Tools - Current state of declarative models.* Tech. Rep. 96-30, Georgia Institute of Technology, Atlanta, 1996.

[Sch97] E. Schlungbaum. *Individual User Interfaces and Model-Based User Interface Software Tools.* In *Intelligent User Interfaces*, pp. 229–232. 1997.

[Sha97] A. Sharp. *Smalltalk by Example: The Developer's Guide.* Mcgraw-Hill, 1997.

[She92] S. Sheppard. *Report on the CHI'91 UIMS Tool Developers' Workshop.* In SIGCHI Bull., vol. 24(1):pp. 28–31, 1992.

[Sin04] D. Sinnig, A. Gaffar, A. Seffah, P. Forbrig. *Patterns, Tools and Models for Interaction Design.* In H. Trætteberg, P. J. Molina, N. J. Nunes (eds.), *MBUI*, vol. 103 of *CEUR Workshop Proceedings.* CEUR-WS.org, 2004.

[sit] *UsiXML: User Interface eXtensible Markup Language.* `http://www.usixml.org/`.

[Sot05] J.-S. Sottet, G. Calvary, J.-M. Favre, J. Coutaz, A. Demeure, L. Balme. *Towards Model Driven Engineering of Plastic User Interfaces.* In J.-M. Bruel (ed.), *MoDELS Satellite Events*, vol. 3844 of *Lecture Notes in Computer Science*, pp. 191–200. Springer, 2005.

[Sot08] J.-S. Sottet, G. Calvary, J.-M. Favre. *Models at Run-time for Sustaining User Interface Plasticity.* In *Workshop on Models@run.time (Models 2008).* 2008.

[Sou03] N. Souchon, J. Vanderdonckt. *A Review of XML-compliant User Interface Description Languages.* In J. A. Jorge, N. J. Nunes, J. F. e Cunha (eds.), *DSV-IS*, vol. 2844 of *Lecture Notes in Computer Science*, pp. 377–391. Springer, 2003.

[Sta08] A. Stanciulescu, J. Vanderdonckt, T. Mens. *Colored Graph Transformation Rules for Model-Driven Engineering of Multi-Target Systems.* In *GraMot'08.* ACM, 2008.

[Suk94] P. N. Sukaviriya, S. Kovacevic, J. D. Foley, B. A. Myers, D. R. Olsen, M. Schneider-Hufschmidt. *Model-Based User Interfaces: What are They and Why Should we Care?* In *ACM Symposium on User Interface Software and Technology*, pp. 133–135. 1994.

[SUN00] SUN. *Scaling the N-tier Architecture.* Tech. rep., SUN Microsystems, 2000.

[Sva05] M. Svahnberg, J. van Gurp, J. Bosch. *A taxonomy of variability realization techniques.* In Softw., Pract. Exper., vol. 35(8):pp. 705–754, 2005.

[Sze95] P. A. Szekely, P. N. Sukaviriya, P. Castells, J. Muthukumarasamy, E. Salcher. *Declarative interface models for user interface construction tools: the MASTERMIND approach.* In *EHCI*, pp. 120–150. 1995.

[Sze96] P. A. Szekely. *Retrospective and Challenges for Model-Based Interface Development.* In *DSV-IS*, pp. 1–27. 1996.

[Tar99] P. L. Tarr, H. Ossher, W. H. Harrison, S. M. Sutton Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In *International Conference on Software Engineering (ICSE)*, pp. 107–119. IEEE Computer Society Press, 1999.

[Tid05] J. Tidwell. *Designing Interfaces.* O'Reilly, 2005.

[Tom00] I. Tomek. *Joy of Smalltalk.* Free on-line book, 2000.

[Van04] J. Vanderdonckt, Q. Limbourg, B. Michotte, L. Bouillon, D. Trevisan, M. F. Florins. *USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces.* In *W3C Workshop on Multimodal Interaction WMI'2004*. Sophia Antipolis, 2004.

[Van05] W. Vanderperren, D. Suvee, M.-A. Cibran, B. De Fraine. *Stateful aspects in JAsCo.* In *Proceedings of Software Composition (SC'05)*, vol. 3628 of *LNCS*. Springer-Verlagger, 2005.

[VdB04] J. Van den Bergh, K. Coninx. *Model-based design of context-sensitive interactive applications: a discussion of notations.* In *TAMODIA*, pp. 43–50. 2004.

[VdB05] J. Van den Bergh, K. Coninx. *Towards modeling context-sensitive interactive applications: the context-sensitive user interface profile (CUP).* In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pp. 87–94. ACM, New York, NY, USA, 2005.

[Ven] R. Vens. *MijnGeld homepage.* Website. `http://www.sepher.nl/`.

[Vis02] VisualWorks. *Gui developer's guide.* Tech. rep., Cincom Systems Inc., 2002.

[Woo95] B. Woolf. *Understanding and using the ValueModel framework in VisualWorks Smalltalk.* In , pp. 467–494, 1995.

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

[Zuk97] J. Zukowski. *JAVA AWT Reference.* O'Reilly, 1997.