A Layered Approach to Dedicated Application Builders Based on Application Frameworks

Patrick Steyaert- Koen De Hondt - Serge Demeyer

Programming Technology Lab
Brussels Free University
Pleinlaan 2, 1050 Brussel, Belgium
E-mail: {prsteyae, kdehondt, sademeye}@vnet3.vub.ac.be
WWW: http://progwww.vub.ac.be/

Marleen De Molder

OO Partners bvba
Romeinse steenweg 464
1853 Strombeek-Bever, Belgium
Fax: +32 2 2677151

ABSTRACT. *In this paper we investigate what is needed to make user interface builders incrementally refinable. The need for dedicated user interface builders is motivated by drawing a parallel with programming language design and object-oriented application frameworks. We show that reflection techniques borrowed from the programming language community can be successfully applied to make user interface builders incrementally refinable.*

KEY WORDS: *User Interfaces, User Interface Builder, Object-oriented Framework, Reflection.*

1. Introduction

Graphical user interfaces for workstation applications are inherently difficult to build. To help programmers create such interfaces, tools are being developed. Those tools range from tool kits (libraries providing primitive building blocks for managing simple widgets, sometimes machine dependent) to user interface management systems (application frameworks and user interface builders supporting higher level user interface concepts). The major goal of those tools is to reduce the effort to build a good user interface and, more importantly, to promote the reuse of application code. Application code is not specific for the problem domain but deals with the interaction of a program with its environment independently from its problem domain. The problem domain is handled by domain code.

In analogy with programming paradigms (e.g. functional and imperative programming), it can be observed that different user interface paradigms (Tesler, 1986) exist. Although the notion of user interface paradigms has not been developed to the same degree as programming paradigms, different user interface paradigms can be tentatively identified. Most popular is the direct manipulation paradigm in which an interface models the real or an imagined world without using intermediate structures such as command languages or menus (Shneiderman, 1983). The exact opposite is the menu driven user interface paradigm. Another popular paradigm, the navigational paradigm, can be found in the area of hypertext systems. Here the idea is to allow users to explore different ideas by supporting the navigation through complex information structures.

As with programming paradigms, user interface paradigms also can be divided into sub-paradigms. For example in (Akscyn, 1991) a distinction is made between domain specific and domain general hypertext systems, whereby domain specific hypertext systems are possibly expressed in particular refinements of the navigational paradigm.

Generally one can say that a particular set of user interface tools is specialised to generate the user interface for applications expressed in a particular paradigm or sub-paradigm. Moreover, it can be observed that some applications can be more easily expressed in one paradigm than others. For example, the navigational paradigm is more suited to express hypertext applications. Similar to programming languages there is a trade-off between generality and the ability to compactly express applications in it. The more general the tool is, the less concise applications can be expressed in it. In principle, dedicated tools minimise, and in some cases eliminate, the need for additional programming. This is because features are absorbed in the system supporting the dedicated paradigm. The absorption makes the dedicated tools less general of course.

The trade-off has lead to a proliferation of user interface tools that cannot co-operate. Instead of defining a user interface tool for each paradigm, we want to be able to define a family of co-

operating user interface tools that support different paradigms. One way to achieve this is to define one customisable user interface builder.
But – and here we come to the central theme of this paper – user interface builders are in most of the cases "closed systems": they generally lack the ability to be adapted or specialised to support new or refined user interface paradigms. A symptom for this illness is the fact that most user interface builders are expressed as 'black box' abstractions (Kiczales, 1992).
This paper discusses techniques for creating open user interface builders. We will investigate the different aspects involved in specialising a general purpose user interface builder based on an application framework to a dedicated user interface builder based on a specialised application framework. Therefore, after covering some basic vocabulary, we will propose a fairly general application framework serving as a basis for our user interface builder. Afterwards, we will show how reflection can be used for specialising user interface builders. Finally our approach will be demonstrated by a well documented experiment.

2. Terminology and Examples

Application frameworks aid in building applications and their user interface by providing a skeleton or abstract implementation that can be reused (Wirfs-Brock, 1990)(Johnson et al., 1988)(Deutsch, 1987). In the object-oriented community, researchers observed that inheritance and late-binding polymorphism are powerful abstraction mechanisms, and that programs expressed in an object-oriented programming language can be reused by incrementally adapting them to different needs. As such, object-oriented techniques are especially useful when building application frameworks. An object-oriented application framework consists of abstract and concrete classes that together form a theory on how to build applications and their user interface. Among the earliest examples of object-oriented application frameworks was the Smalltalk Model/View/Controller framework (Krasner et al., 1988) (Goldberg et al., 1989)(Lalonde et al., 1991). Other examples are MacApp (Schmucker, 1986) and InterViews (Linton et al., 1989).
Since application frameworks are expressed as object-oriented skeleton programs, standard object-oriented techniques such as refinement by inheritance can be applied to it. A skeleton can be incrementally modified into a new, more specialised skeleton. An example can be found in (Meyrowitz, 1986) where MacApp, a fairly general application framework, is refined to Intermedia, an application framework for hypermedia applications.
User interface management systems (UIMS) promote visual user interface design and development to minimise the need for conventional programming. A *user interface builder* (UIB) is a particular kind of UIMS. Conventionally UIBs are interactive meta-programs that allow visual painting of user interfaces and generation of application code. The generated programs are skeleton implementations to which the programmer must further add the user interface logic and domain logic. In the ideal case where the target language is object-oriented, the generated code can fit in some application framework that can be reused incrementally to build an application. In other cases the application code and domain code must be added, e.g. by text editing the generated skeleton. In all cases the need for conventional programming is minimised, since the bulk of the interface management is absorbed in the underlying user interface model.
To illustrate these ideas, we will discuss two examples of user interface builders: HyperCard and VisualWorks. The former is a dedicated purpose user interface builder, promoting an iterative development process where no programming is necessary to build running applications. The latter is general purpose in nature, supports an iterative development process, and requires programming to build an application. Both of the examples will be used later on in this paper.

HyperCard (Goodman, 1987) is a kind of hypertext. Conceptually, a HyperCard application is a stack of cards. Each card contains some information and links to other cards in the same or other stacks. The information on the cards is shown using text and graphics. The links to other cards are presented as buttons, typically complete with an icon representing the destination card. A user of HyperCard browses the cards of a stack using the link buttons. Only one card of a stack is displayed at a time. Clicking a link button results in the display of the destination card.
HyperCard not only lets a user browse prefabricated stacks (browsing and typing level), it also allows the creation of new stacks (authoring level). On the authoring level, the stack developer designs cards using the small set of HyperCard-dedicated interface components. Text fields and

pictures are used to display information. Link buttons are used to link cards. After painting an interface component, its contents is to be specified in order to complete its appearance. In case of a link button, its destination card must be specified by selecting it.
At any time the stack developer can switch from painting to browsing and back to try out the new cards and to fine-tune his design. This iterative development ensures a very smooth transition from development to a running stack. For simple information-oriented stacks, the developer does not need to write a single line of code. Only the destination cards of the link buttons need to be specified. However, when a stack not only has an informational nature, but also has to exhibit a more active behaviour, the stack designer has to develop cards on the scripting level, on which programming in the dedicated language HyperTalk is supported. For example, a button can be assigned another behaviour than simple card linking.

VisualWorks[1] (VisualWorks, 1992) is a development tool used in conjunction with the ObjectWorks\Smalltalk environment. We refer to section 3.3 for a discussion on the concepts used in VisualWorks; here we will illustrate the development process with the construction of an example application.
Imagine a dialog window to manipulate an integer: a possible interface might consist of a text field and two buttons. In the text field the user is allowed to edit the value using the keyboard; the buttons are used for incrementing or decrementing the value. Figure 1 shows how such an interface may look like.

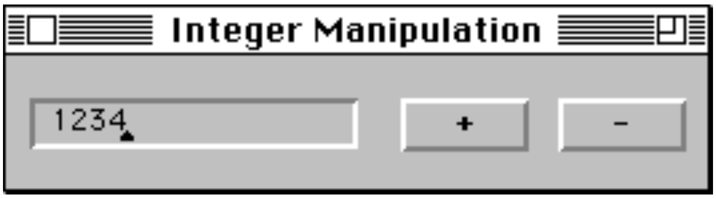

Figure 1: a running application to manipulate an integer

To create this interface, the interface designer opens a new canvas (an editor representation for the interface being developed) and shapes it using the painter. He drags visual components (here two buttons and one text field) from a palette onto the canvas and positions them appropriately. Afterwards he uses the properties editor to define properties of these components.
After stepping through this procedure, he has created the user interface for the application without writing a single line of code. But to incorporate the user interface logic, programming is required. First, a class must be created as a repository for the code. Second, the interface designer uses VisualWorks tools to generate code for every visual component on the canvas. Regular interface development forces designers to extend the application framework with user interface logic afterwards.
In this example, the VisualWorks tools are able to create an instance variable holding the text entered in the text field; the appropriate initialisation and accessor methods are generated as well. For every button an empty action method is generated, which must be completed by the interface designer. Here is where the user interface logic is defined (i.e. the fact that when pressing a button, the value in the text field is incremented or decremented).
As with normal Smalltalk development, it is possible to build and test applications incrementally. In this example, one can build a window with one text field, test it, and install the buttons afterwards. The incremental development idea is not completely supported though, since the VisualWorks system is unaware of modifications to generated code.

3. Related Work

As explained before, user interface management systems are software tools that enable designers to create a complete and working user interface and minimise the coding effort.
The main concerns of UIMSs are listed by Alan Dix (Dix et al., 1993 - p. 353):"

---

1 VisualWorks, ObjectWorks and ObjectWorks\Smalltalk are trademarks of ParcPlace Systems Inc.

- a conceptual architecture for the structure of interactive systems which concentrates on a separation between application semantics and presentation,
- techniques for implementing a separated application and presentation while preserving the intended connection between them,
- support techniques for managing, implementing and evaluating a run-time interactive environment."

Main reasons for using UIMSs are portability and reusability, arguments we recognise as advocating for application frameworks too.

In the following paragraphs we will discuss some important UIMSs and match them to the above criteria. Afterwards we will tentatively define a purified application framework and associated UIB.

3.1 The Model-View-Controller Framework

The Model-View-Controller (MVC) framework (Krasner et al., 1988)(Goldberg et al., 1989)(Lalonde et al., 1991) can not be omitted from this overview, as it is generally regarded as being one of the first object-oriented frameworks (Johnson et al., 1988)(Johnson et al., 1991). Its main contribution was the distinction between three important roles (namely data, output and input) that should be reflected in the design of a user interface framework. Those roles were reflected in three abstract superclasses — Model, View, Controller — composing the heart of the framework. It also provided a dependency mechanism between objects: a construction for change propagation (Weinand et al., 1988) which is necessary when developing highly interactive software. As such it was used in building the graphical user interface for the Smalltalk programming environment.

In (Tesler, 1986) it is argued that the MVC framework is very broad, and can be used to explore different user interface paradigms. Nevertheless, it needs more refinement to allow for actual code reuse. In the ObjectWorks\Smalltalk 4.1 (ObjectWorks, 1992) class library this is accomplished with a group of 'pluggable views'. In practice we feel that this is not enough to call it a real UIMS: there is still too much burden left on the programmer (e.g. the positioning of the visual elements). Moreover, it is hard to scale up MVC applications since all of the synchronisation between the visual elements is encoded in the model: a better separation of responsibilities is needed. These are well known problems and they were addressed with the introduction of VisualWorks (section 3.3).

3.2 The MoDE Framework

One of the first extensions of the MVC framework was the MoDE framework (Shan, 1990). The motivation for the experiment was the observation that the MVC concepts provide convenient divisions at the abstract level, but maintaining this division on the implementation level is much harder. This hinders the reuse of software components and produces awkward inheritance structures.

MoDE (Mode Development Environment) is what we call a UIB. It consists of a direct manipulation interface editor (the Mode Composer) on top of (and created with) an object-oriented framework. The basic building block of the framework is a mode, which is responsible for managing a certain area on the screen. A mode is defined by its appearance, its semantics and its interaction: three orthogonal axes in the user interface component space, each of them corresponding with different classes in the framework. The interaction between the different modes is coded in the *computing component*. MoDE has been used to develop several user interfaces supporting different user interface paradigms (direct manipulation, navigational, ...). The framework is easy to extend for experienced programmers. The composer promotes the reuse of modes: newly created modes may be pasted in a library for later application.

Our main objection against MoDE is the vague definition of the "computing component". This encourages the intertwining of user interface logic with domain logic (constraints independent of the user interface) and obstructs the reuse of software components.

3.3 *VisualWorks*

The problems of the class library provided with ObjectWorks\Smalltalk (see section 3.1) were tackled with the introduction of VisualWorks (VisualWorks, 1992). The major improvement came with the direct manipulation interface editor. Similar to the NeXT interface builder (Webster, 1989)(Shneiderman, 1992) it provides the interface designer with a palette of widgets that may be pasted on a canvas (an editor representation for the actual window). Each of these widgets (visual components) communicates with the underlying model through an *aspect*, which is responsible for the user interface behaviour of a certain piece of information[2]. To control the interrelationships between those aspects, the notions of *application model* and *domain model* were introduced: the former being responsible for the user interface, the latter for the data consistency.

There exist other commercial products providing graphical interface editors for Smalltalk, but none of these possesses fundamental improvements compared to VisualWorks. In fact they all lack an important property we feel as being essential for the success of user interface builders: extensibility. Indeed it is possible to expand the set of available widgets, but this is not enough. First of all because the process is too difficult (it is poorly documented and requires some tricky hacks). Secondly because incorporating new interface paradigms takes a lot more than the ability to include new widgets. For example, when we want to simulate HyperCard, we would like to include a "home button" into the palette of available widgets. To do this we need a way to provide semantic information for the widget. We refer the reader to the last section of the paper for a detailed discussion on the subject.

What is needed is a well defined architecture and an open and extensible implementation. And here we touch upon another severe drawback of the VisualWorks system: the lack of a well-defined application framework. A symptom for this is the fact that there are no classes for important concepts like "DomainModel" and "Aspect".

3.4 *Conclusion*

Over the years many developers used object-oriented frameworks as the kernel of a UIMS. We mention HotDraw (Johnson, 1992) and Tigre (Tigre, 1991) as additional UIMSs for Smalltalk. Other languages served as implementation vehicle as well: MacApp (Schmucker, 1986) for C; ET++ (Weinand et al., 88), Vamp (Ferrel et al., 1989) in C++; Picasso (Konstan et al., 1991) for CLOS.

It is our opinion that all of today's UIMSs fail in making a clear separation between user interface logic and domain logic. This is necessary for an orthogonal design enabling code reuse. Moreover, when providing the means for extending the UIMS, they are biased towards user interface details. Adding new kinds of widgets is feasible, but incorporating semantic information needed to assimilate interface paradigms is nearly impossible.

In the next section we will see how we can improve on these shortcomings by introducing a new architecture for user interface management systems.

4. Building Application Builders

As explained before, user interface management systems promote visual user interface design and the reuse of visual components in user interfaces. Some of these systems even permit experienced users to extend the library of widgets. In spite of all these powerful properties, we feel that too much attention is directed towards the visual characteristics and not enough to the underlying user interface model.

We claim that a higher level of reuse and extension is needed to incorporate user interface paradigms into a user interface management system. We will call a system that enables such high level manipulation an *application builder*.

In what follows we will try to explain how such a system might be conceived. Afterwards we will guide the reader through an experiment demonstrating the philosophy of an application builder.

---

2 The notion of an aspect resembles in many ways the notion of a mode in MoDE (see section 3.2).

4.1 An Application Framework Sustaining Application Builders

In order to successfully incorporate user interface paradigms into a user interface management system, we propose a scheme of incremental extensions on an underlying framework. We will start with the definition of the concepts serving as the foundation for our UIMS.
It is not a claim of this paper to propose the ultimate UIMS framework. As a matter of fact, we don't think such a framework exists. Rather, we want to argue that the use of a framework allows the construction of an "open" user interface builder. Moreover, reflection makes it possible to migrate seamlessly from the user interface design level to the paradigm design level.
For easy refinement of a framework, a set of orthogonal concepts is essential (Johnson et al., 1988). We adopt the terminology of VisualWorks (see 3.3), mainly because we like its clear distinction between the application model and domain model. We purify these concepts to fit them into the framework.
In figure 2 the foundation of the framework is depicted. The core of the model consists of three basic concepts: user interface component, application model and domain model. An auxiliary concept, aspect, is needed to separate the domain model from the user interface component. The relations between the four concepts are display, notify and control.

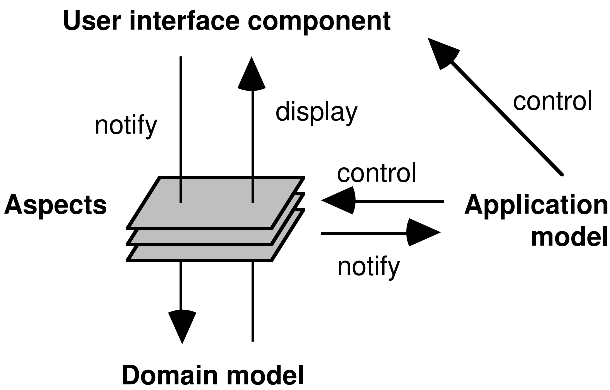

Figure 2: the framework architecture

A *domain model* models the overall functionality of the problem domain and maintains user interface independent constraints. From the viewpoint of the user interface components, its main goal is to serve as a storage for the information to be displayed. Features like printing, persistence (database storage) and network communication are included in the domain model as well. There can exist several application models for one domain model.
An *aspect* is a container for a reference to a piece of information (supplied by the domain model) that is to be represented on the screen and can be modified by the user. Such information may be simple (e.g. strings, numbers, dates) or complex (e.g. lists, matrices). The main function for an aspect is to interpret operations from user interface components and translate them into operations on the domain model. As such, aspects can be layered on top of each other to encapsulate user interface specific operations from the domain model.
A *user interface component* controls the display and the user interaction of a particular piece of information. The information is supplied by the domain model, but a series of aspects will be used to hide implementation details, e.g. the message that is needed to retrieve the information from the domain model, or the last choice from a pop-up menu. Among the responsibilities of a user interface component are properties like colour, font and position (display), and functions like mouse tracking and keyboard control (user interaction). Note that every interface component has exactly one aspect, but that one aspect can be shared between several components. Normally an interface component will not stand on itself: it is grouped with other interface components to act as a whole.
An *application model* manages the global behaviour of such a group of interface components. It is responsible for the user interface logic and controls user interface behaviour like: When should what information be displayed ? What operations affect the information and how should the display be updated ? As such, an application model is an aggregation of aspects that are to be

considered as a whole and interact with each other. Actually, almost all of the programming effort that goes into an application model is specifying the interaction between the different aspects. Readers should be aware that the same application model can be reused on different domain models.

It is recognised that user interface tools require a mechanism to synchronise the different interface components (Krasner et al., 1988)(Weinand et al., 1988). That is precisely the motivation for the relations between the basic concepts.
After setting up a complete data structure, the system starts with an initial display. This is accomplished by activating the user interface components for all aspects contained in the application model. The user interface component requests the aspect for the information to display, the aspect translates this request in the appropriate operations on the domain model (or on the next aspect). This explains the *control* relation from the application model to the user interface component and the *display* relation from the domain model to the interface component.
When the user activates a user interface component (by clicking a button, selecting an item in a list or a pop-up menu, typing some characters in a text field, etc.) it applies a *notification* operation on the domain model (through the aspect) which takes the appropriate actions. Most of the aspects also *notify*[3] the application model. The application model might decide to alter interface components (e.g. disabling a text field) or to modify the domain model through other aspects. This explains the two *notify* relations (from the interface component to the domain model and from the aspect to the application model) and both of the *control* relations (from the application model to the interface component and to the aspect).

4.2 An Example

In this section we will illustrate the architecture of our framework with the integer manipulating application from section 2.
The domain model for such an application is a simple object with one instance variable being an integer. Let us call this variable 'number' (with accessor methods `number` and `number:`) and the class representing the domain model 'UserObject'. We define three user interface components for this application: a text field and two buttons. The text field should allow numeric characters only, the label of the increment button is a plus sign and the label of the decrement button is a minus sign.

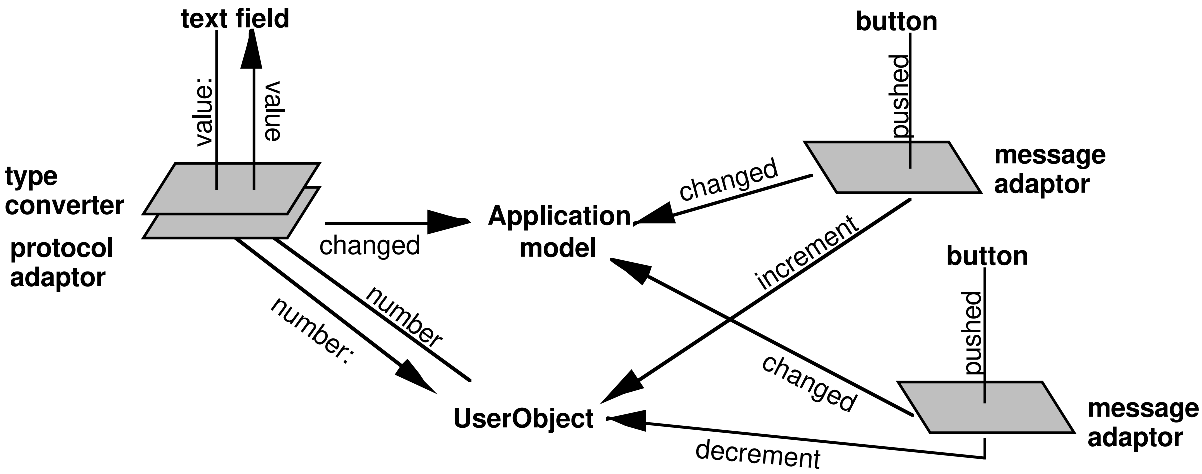

Figure 3 : the structure behind the number application

Some aspects are needed to glue these user interface components together with the application and domain model (see figure 3) :

---

3 The mechanism is similar to the dependency mechanism of Smalltalk (see section 3.1).

- A protocol adaptor to translate the protocol of the text field (i.e. `value` / `value:`) into the protocol of the domain model (i.e. `number` / `number:`) and to notify the application model using the `changed` message.
- A type converter to convert the type of the text field (a string) into the type of the domain model (an integer).
- A message adaptor to translate the protocol of the buttons (i.e. `pushed`) into the protocol of the domain model (i.e. `increment` / `decrement`) and to notify the application model using the `changed` message.

4.3 User Interface Builder with an Open Implementation + Reflection = Application Builder

In our view a user interface builder consists of a range of tools. The *painter* is the tool to create a user interface layout. It consists of a canvas and a palette of visual components. Aspects are chosen from the aspect palette and associated with visual components on the canvas. Visual properties and aspects are modified with the *visual properties editor* and the *aspect editor* respectively. User interface logic is generated by the *definer*, and modified using the *code browser* or other conventional coding tools.
A particular configuration of tools determines how well the UIB fits into a certain user interface paradigm. Adapting the UIB to a new interface paradigm amounts to the reconfiguration of this set of tools. As already argued in the introduction, today's UIBs do not support this reconfiguration, because they are expressed as 'black box' abstractions (Kiczales, 1992).
'Open implementations' (Rao, 1991) do support this reconfiguration of tools by providing a meta-level interface (Rao, 1991)(Kiczales, 1992)(Steyaert, 1994) besides the traditional base-level interface. The latter is the interface to the basic system's functionality. The former is the interface to the system's implementation and reveals aspects of how the base-level interface is implemented.
A UIB with an open implementation is a UIB with a meta-level interface to adapt the configuration of the tools it consists of. For example, to adapt the painter tool the meta-level interface supplies the methods `installComponentPalette:`, `installAspectPalette:` and `installDefaultCanvas:` for the installation of custom palettes and custom default canvasses.
Only providing a meta-level interface has the severe drawback that the UIB modifications have to be hand-coded. This means that an interface designer who finds the UIB not suited for his target applications must use a completely different set of tools to adapt the UIB to his needs. To reconfigure the UIB, he must leave the UIB environment, hand-code the modifications, install them into the UIB-framework and return to the UIB environment. As a consequence only skilled users would be able to customise the UIB. Moreover, switching levels gives rise to certain 'paradigm mismatches', e.g. inconsistencies between hand-coded specifications and specifications generated by the UIB.
New tools need not be hand-coded as is typically the case for their built-in counterparts. Since they have a user interface, they can be created by a UIB. A UIB with an open implementation capable of reconfiguring itself is called reflective. It uses its meta-level interface for its own specialisation. A reflective UIB can be used for both the construction of traditional applications and the specialisation of itself. Reflective UIBs enable incorporation of user interface paradigms without leaving the UIB environment. This is precisely the higher level reuse and extension aimed at in the introduction of section 4. Thus, a UIB with an open implementation and reflection is an application builder.

The reflective nature of an application builder introduces layers of abstraction in the user interface building process. At any level, the application builder is used to refine the framework introduced in section 4.1. At the bottom layer the basic application builder is used; at any level above all refinements of the previous levels can be reused. This is illustrated in the next section.

5. Experiment : a HyperCard-like Application Using Specialised UI Components

To substantiate the idea of a reflective application builder, experiments were carried out in VisualWorks. Natively VisualWorks does not support customisation. Therefore it was extended with reflective facilities. Moreover VisualWorks uses a slightly different application framework than the one proposed in this text. We will point out the differences when needed.

The experiment described here uses specialised user interface components in order to build HyperCard-like applications. Figure 4 shows a card of the target application. In the remainder we will call the HyperCard-like application 'Card'. We aimed to create user interface components and one application model that are so dedicated to the Card application that the Card application programmer need not write a single line of user interface logic. The dedication lies in the fact that all of the features needed for the definition of the user interface logic can be absorbed in the user interface components.

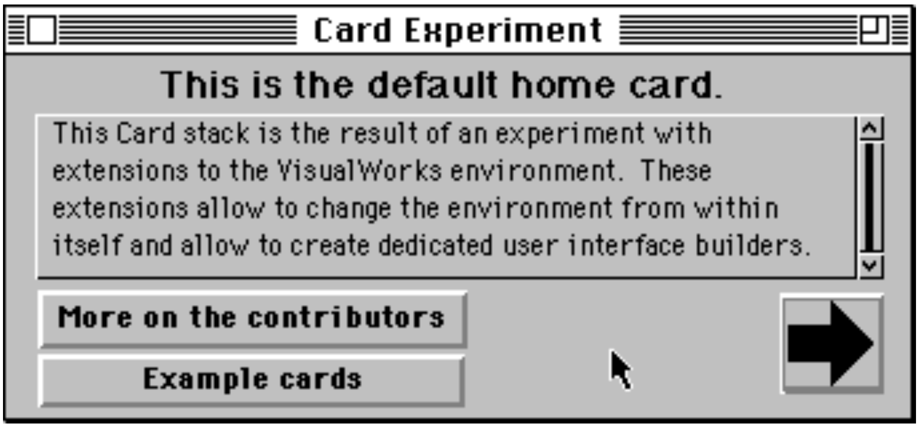

Figure 4 : an example card

First a description will be given of how the Card application can be designed with standard VisualWorks. VisualWorks' shortcomings will be discussed as well. Then some light will be shed on how VisualWorks was extended to turn it into a reflective variant. Finally, the development of the Card application using the reflective variant of VisualWorks will be discussed.

5.1 Description of the User Interface Components and the Application Model

The description of HyperCard was given in section 2. For simplicity, we will restrict the information to be textual. Conceptually, five link button types can be distinguished: the general link button and four special link buttons: the home button, the previous button, the next button and the return button. The general link button links with an arbitrary card in the stack. The home button connects with the first card in the stack. The previous and the next button are used to follow a sequential path through the stack and the return button jumps to the last visited card. These five button types will be made explicit in the Card application.
Labels, text fields and buttons are placed on a card. Cards are represented as complete interfaces. In order not to get a proliferation of non-reusable application models representing cards, all cards share the same application model, being the Card application. There is no explicit domain model (i.e. a stack); the order of the cards is determined by explicit linking through the previous button and the next button.

5.2 Designing a HyperCard-like Application with Standard VisualWorks

When using standard VisualWorks components, the different Card buttons can be implemented using the standard action button. The properties of an action button are specified using the properties dialog shown in figure 5. The contents of the 'Action:' field is the name of the message that will be sent to the application model when the action button is clicked.

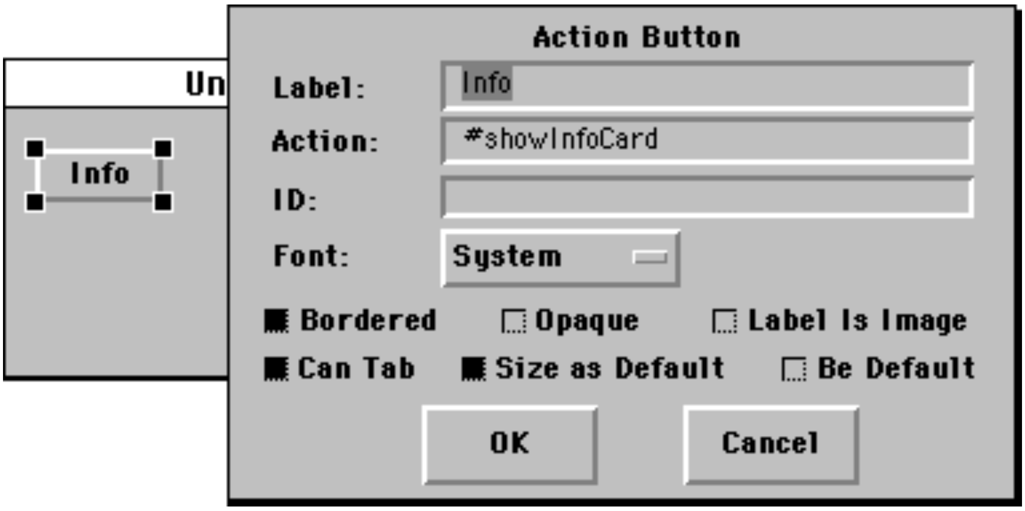

Figure 5 : standard action button properties dialog

In order to make action buttons behave as link buttons, the application model using the action buttons has to provide one action method per action button, each different in the destination card name used (see figure 6). Since our intention was to have only one application model, the Card application model has as many implemented action methods as there are action buttons on all the cards of the stack. Note that the designer of a stack has to implement all the action methods by hand to give each button its specific behaviour.[4]

Figure 6 : an implementation of Card buttons directly based on action buttons

5.3 Arguments for Customisation of the UIB

It is clear that an extra abstraction level is needed to eliminate the multiple action methods. The kernel of a Card application is an application model for navigating through a stack of cards. Therefore it should have a very simple protocol. It should only respond to `gotoCard:` messages to display another card. In contrast with general purpose action buttons, where the different action methods are unary messages, this message takes a parameter: the card to go to when the button is clicked. The destination card of a link button is encapsulated as a variable in the link button aspect. In figure 7 the link button is represented in terms of our conceptual model of section 4.1.

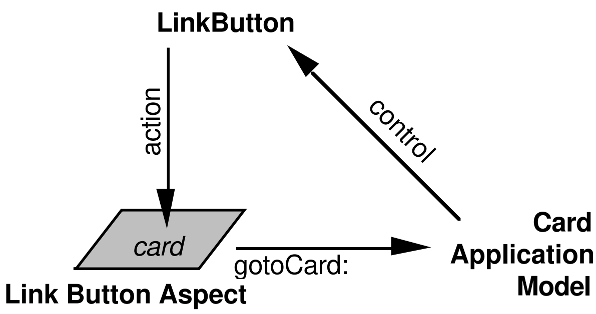

Figure 7 : conceptual representation of a dedicated link button in an abstract application model

4 Having several application models, i.e. one application model per card, would not inhibit the hand-coding of the action methods.

The extra abstraction level can be realised by means of a dedicated UIB and a dedicated Card application model. Instead of using standard action buttons, dedicated link buttons are painted on a canvas. They have dedicated properties dialogs, as shown in figure 8. Link buttons now have a *card* property which specifies the name of the destination card. The name supplied in the 'Card:' field is the actual argument for the `gotoCard:` method of the Card application model. Since all link button aspects use the same method and the method is part of the Card application model's protocol, the designer of a stack does not have to write a single line of code.

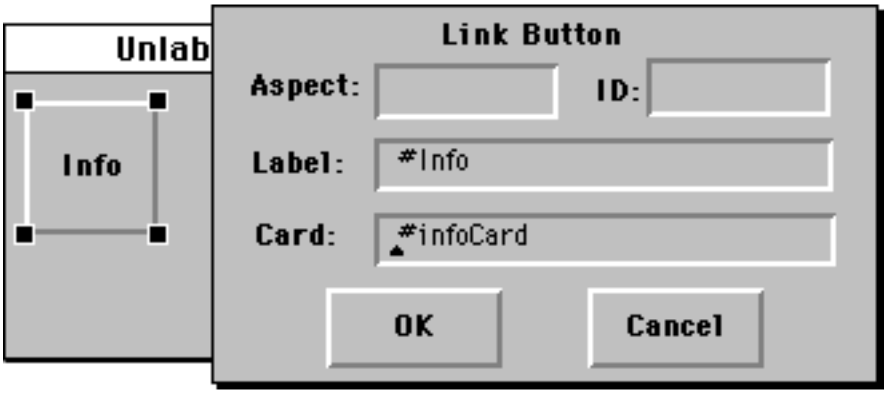

Figure 8 : the dedicated link button properties dialog

Note that the properties editor shown in figure 8 mixes visual properties (e.g. the label of the button) and aspect properties (e.g. the card name). This follows from the fact that VisualWorks does not make an as clean separation between aspects and application models as our framework does. In an ideal UIB there would exist a visual properties editor and an aspect editor to reflect this separation of concepts.

5.4 Customisation of the UIB

To customise the UIB three steps must be taken:

1. creation of dedicated application models
2. creation of properties dialogs for the dedicated application models
3. creation of a palette containing the dedicated application models

The standard VisualWorks UIB does not support steps 2 and 3. Therefore the standard UIB was turned into a reflective variant.
A palette containing user interface components for painting palettes (i.e. palette buttons) was created using the VisualWorks UIB. The palette application model, not present in VisualWorks, and the palette button application models were hand-coded. Then the standard UIB was parameterised with the palette to be used. When opened with the palette for painting palettes, the UIB can now be used to paint dedicated palettes which, in turn, can be used in conjunction with the UIB to paint dedicated user interfaces. The UIB was extended to be able to install a canvas as custom properties dialog, and modified to open the appropriate custom dialog on a dedicated application model.
The UIB is now reflective and will be used to design the HyperCard-like application.

5.4.1 Step 1 : Creation of Dedicated Application Models

The dedicated application models corresponding to the five button types (i.e. general link button, home button, previous button, next button and return button) and the text field are defined using conventional programming, i.e. with a browser. Their user interfaces are specified using the UIB with the standard palette.
The definition of a customised link button aspect is straightforward and depicted below. It is important to mention that this definition is not VisualWorks source code, but rather a definition that fits into the conceptual framework.

```
class LinkButtonAspect extends AbstractAspect
instance variables
    card
methods
    card
        return card
    card: aCard
        card := aCard
    action
        self changed: #gotoCard: with: self card
endclass
```

The link button aspect contains its destination card name. When triggered by an action method, it notifies its application model via the dependency mechanism to go to the destination card. Notice that in the philosophy of section 4.3 the `card` and `card:` methods belong to the meta-level interface and the action method is part of the base-level interface of the link button. The explanation for this is straightforward: the `card` messages are sent by the aspect editor for the definition of the aspect, the `action` message is sent when the button is triggered by user interaction.

5.4.2 Step 2 : Creation of Properties Dialogs

Using the reflective UIB, the properties dialogs of the custom components are specified. The properties dialog for link buttons was depicted in figure 8. The VisualWorks properties dialogs allow editing of aspects as well as visual properties. In our conceptual framework only the aspect editor part is of concern. The responsibility of the aspect editor is to fill in the destination card of the link button aspect. As shown in figure 9, the link button aspect editor is an application which must have the above defined link button aspect as domain model. This is what makes the dedicated UIB special: it allows the installation of aspects, e.g. link buttons, as domain model for aspect editors, e.g. link button aspect editors.

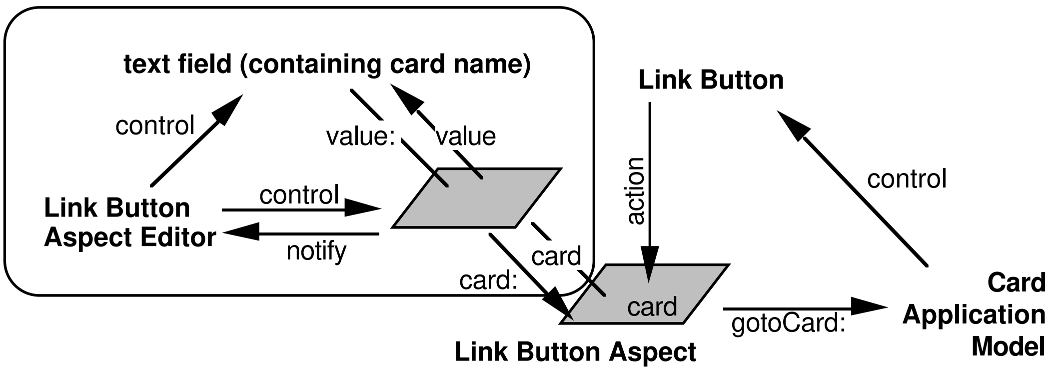

Figure 9 : conceptual representation of the link button aspect editor

5.4.3 Step 3 : Creation of the Dedicated Palette

With the reflective UIB the dedicated palette for Card applications is defined. For all palette buttons the appropriate icons and the target user interface components are specified with the properties editor. In figure 10 on the left the finished custom palette is shown.

5.5 Building an Example Application

Now we are ready to paint canvasses using the new specialised components. For each card a canvas is painted and installed under a different name. The cards are linked through the 'Card' property of the buttons. Figure 10 shows how the dedicated palette is used to paint dedicated Card components on a canvas. Note that the home button aspect editor, as shown, has no card field because the destination card for home buttons is implicit.

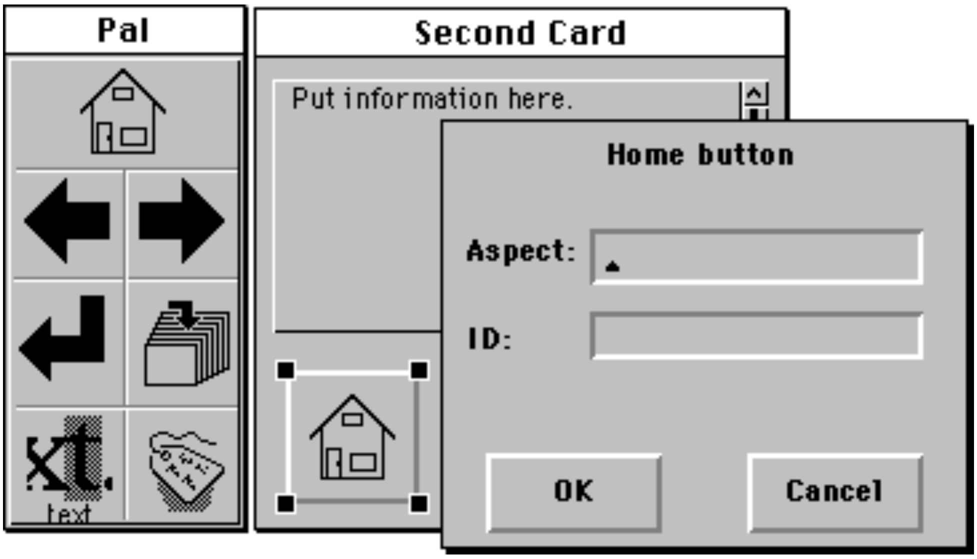

Figure 10 : the Card application builder in action

For a screen dump of a complete card built with the dedicated Card application builder we refer the reader to figure 4.

6. Conclusions

Over the last years we have seen a proliferation of interface tools. Such tools range from simple machine dependent tool kits to full-blown direct manipulation environments. On the other hand, in the object-oriented research community, application frameworks have been successfully applied on diverse problem domains. We feel that for the next generation of user interface builders much can be learned from the experiences gained in application framework research. Especially the concept of incremental refinement seems appropriate.

In this paper we have shown that application frameworks and reflection are essential for the construction of full-blown specialisable user interface builders, because they support such an incremental refinement through a layered approach. Experiments sustain this viewpoint. The experiment described in the paper is a good example of the absorption of the HyperCard user interface paradigm into an application builder. It shows clearly that reflection makes it possible to migrate seamlessly from the user interface design level to the paradigm design level.

Emphasis has been put on the use of reflection, rather than on the technical aspects of making a user interface builder reflective. Although the latter is a very interesting topic, it is out of the scope of this paper.

References

(Akscyn, 1991) R. M. Akscyn. *Design Tradeoffs for Advanced Hypertext Technology*. Tutorial Notes, Third ACM Conference on Hypertext, December 91.

(Cox, 1986) B. Cox. *Panel: User Interface Frameworks*. In Proceedings of OOPSLA’86 conference, pp. 497-501, printed as SIGPLAN Notices, 21(11), 1986.

(Deutsch, 1987) L. P. Deutsch. *Levels of Reuse in the Smalltalk-80 Programming System*. In Peter Freeman (Ed.) Tutorial: Software Reusability, IEEE Computer Society Press, 1987.

(Dix et al., 1993) A. Dix, J. Finlay, R. Beale. *Human-Computer Interaction*. Addison Wesley, 1993.

(Ferrel et al., 1989) P. J. Ferrel, R. F. Meyer: Vamp. *The Aldus Application Framework.* Proceedings of OOPSLA’89 conference, p. 185-189, ACM Press, October 1989

(Goldberg et al., 1989) A. Goldberg, and D. Robson. *Smalltalk-80, The Language*. Addison-Wesley Publishing Company, Reading Massachusetts, 1989.

(Goodman, 1987) D. Goodman. *The Complete HyperCard Handbook*. Bantam Computer Books, September 1987.

(Johnson et al., 1988) R. E. Johnson, B. Foote. *Designing Reusable Classes*. Journal of Object-Oriented Programming, 1(2), pp. 22-35, 1988.

(Johnson et al., 1991) R. E. Johnson, Vincent F. Russo. Reusing *Object-Oriented Design*. University of Illinois tech report UIUCDCS 91-1696,1991.

(Johnson, 1992) R. E. Johnson. *Documenting Frameworks using Patterns*. Proceedings of OOPSLA'92 conference, p. 63-76, SIGPLAN Notices, ACM Press, October 1992

(Kiczales, 1992) G. Kiczales. *Towards a New Model of Abstraction in the Engineering of Software*. In Proceedings of IMSA'92, International Workshop on Reflection and Meta-Level Architectures, pp. 1-11, November 1992.

(Konstan et al., 1991) J. A. Konstan, L. A. Rowe. *Developing a GUIDE Using Object-Oriented Programming*. Proceedings of OOPSLA'91 conference, p. 75-88, SIGPLAN Notices, ACM Press, November 1991

(Krasner et al., 1988) G. E. Krasner, S. T. Pope. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. Journal of Object Oriented Programming, August 1988, p. 26-49.

(Lalonde et al., 1991) W. R. Lalonde, J. R. Pugh. *Inside Smalltalk - Volume II*. Prentice Hall 1991.

(Linton et al., 1989) M. A. Linton, J. M. Vlissides, P. R. Calder. *Composing User Interfaces with InterViews*. IEEE Computer, p. 8-22, February 1989

(Meyrowitz, 1986) N. Meyrowitz. Intermedia: *The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework*. In Proceedings of OOPSLA'86 conference, pp. 186-201, printed as SIGPLAN Notices, 21(11), 1986.

(ObjectWorks, 1992) ParcPlace Systems. *Objectworks\Smalltalk Release 4.1 User's Guide*. ParcPlace Systems 1992.

(Rao, 1991) R. Rao. *Implementational Reflection in Silica*. In ECOOP'91 Proceedings, Lecture Notes in Computer Science, P. America (Ed.), pp. 251-267, Springer-Verlag, 1991.

(Schmucker, 1986) K. J. Schmucker. *Object-Oriented Programming for the Macintosch*. Hayden Book Company, 1986.

(Shan, 1990) Y. P. Shan. *MoDE: An Object-Oriented User Interface Development Environment Based on the Concept of Mode*. PhD Thesis, University of North Carolina at Chapel Hill - Department of Computer Science, 1990.

(Shneiderman, 1983) B. Shneiderman. *Direct Manipulation: A Step Beyond Programming Languages*. IEEE Computer, August 1983.

(Shneiderman, 1992) B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison Wesley 1992.

(Steyaert, 1994) P. Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. PhD Thesis, Vrije Universiteit Brussel, 1994.

(Tesler, 1986) L. Tesler, position statement in (Cox, 1986).

(Tigre, 1991) Tigre Object Systems. *The Tigre Programming Environment, User's Guide, Version 1.6*. Tigre Object Systems, 1991

(VisualWorks, 1992) ParcPlace Systems. *VisualWorks release 1.0 User's Guide*. ParcPlace Systems 1992.

(Webster, 1989) B. F. Webster. *The NeXT Book*. Addison-Wesley 1989.

(Weinand et al., 1988) A. Weinand, E. Gamma, and R. Marty. *ET++: an object-oriented application framework in C++*. In Proceedings of OOPSLA'88, pp. 46-57, November 1988, printed as SIGPLAN Notices, 23(11).

(Wirfs-Brock, 1990) Allen Wirfs-Brock. *Panel: Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks*. SIGPLAN Notices Special Issue OOPSLA-ECOOP'90 Addendum to the Proceedings (Jerry L. Archibald and K.C. Burgess Yakemovic eds.), pp. 19-24, 1990.