Reflective Application Builders

Patrick Steyaert[*] Koen De Hondt[†]

Serge Demeyer[‡] Niels Boyen[§]

January 26, 1996

Abstract

Current visual application builders and application frameworks do not live up to their expectations of rapid application development or non-programming-expert application development. They fall short when compared to component-oriented development environments in which applications are built with components that have a strong affinity with the problem domain (i.e. being domain-specific). Although the latter environments are very powerful, they are hard to build and, in general, do not allow much variation in the problem domain that is covered. In this paper we show how this apparent conflict between generality and domain specificity can be overcome by considering application building itself as the problem domain. This naturally leads to the notion of a reflective application builder, i.e. an application framework—application builder pair that incorporates all the tools for the visual construction of (domain-specific) application builders.

[*]Programming Technology Lab, Computer Science Department, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium; www: http://progwww.vub.ac.be/; email: prsteyae@vnet3.vub.ac.be

[†]Same address; email: kdehondt@vnet3.vub.ac.be

[‡]Same address; email: sademeye@vnet3.vub.ac.be

[§]Same address; email: nlboyen@vub.ac.be

1 Introduction

A recent trend in software development is the fostering of component-oriented software development in response to the claimed failing of the object-oriented paradigm [Ude94]. However, comparing component-oriented and object-oriented software development is like comparing apples and oranges. Both target the highly desired goal of reuse, but the means to accomplish this goal are complementary rather than mutual exclusive:

Strengths of componentware: end-user oriented, visual programming, rapid application development, large libraries of predefined components.

Strengths of object-orientation: general purpose, solid software engineering capacity, application framework development, extendibility of class libraries, availability of application builders.

In our experience component-oriented environments suffer from their closed nature, which limits the applicability of componentware. Conventionally, components can only be composed in an inflexible framework. Thus, users rapidly encounter the limits of the system, due to a lack of solid software engineering techniques offered. This is reinforced by the wide gap between composing components into an application and the development of new components or adapting the environment.

We are working on an application development environment (ApplFLab, or Application Framework Laboratory) [SDHDDM95] where solid object-oriented software engineering techniques go hand in hand with rapid application development by visual composition. This environment supports the construction of object-oriented frameworks together with the tools for visual composition of components in the constructed frameworks. This constitutes a true domain-specific software architecture: rapid application development through visual composition by domain experts (not necessarily programmers) based on a soundly engineered framework architecture produced by software engineers. ApplFLab has the specific advantage that the domain experts and the software engineers work in the same medium. The integration is achieved by combining reflection techniques [Rao91] with object-oriented frameworks and graphical application builders.

In this paper, we only focus on the visual composition of the components of the constructed frameworks, not the full ApplFLab development environment. We investigate the different aspects involved in specialising a general purpose application builder based on an application framework in contrast to a dedicated application builder based on a specialised application framework. After covering some basic terminology, we will discuss a general application framework serving as a basis for our application builder. Afterwards, we will show how reflection can be used for specialising application builders. Finally, we will demonstrate that a reflective application builder can be used as a tool for the construction of new components for domain-specific frameworks and for the visual composition of the different objects in those frameworks.

2 Terminology

Application frameworks aid in building applications and their user interface by providing a skeleton or abstract implementation that can be reused [WB90, JF88, Deu87]. In the object-oriented community, researchers observed that inheritance and late-binding polymorphism are powerful 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 [KP88, GR89, LP91]. Other examples are MacApp [Sch86] and InterViews [LVC89].

Since application frameworks are expressed as object-oriented skeleton, 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 [Mey86] 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. An *application builder* is a particular kind of UIMS. Conventionally, application builders are interactive programs that allow visual composition of user interfaces and generation of application code. They are metaprograms since they generate skeleton programs to which the programmer must further add the user interface logic and domain logic. In object-oriented languages, the generated code fits in an application framework that can be reused incrementally to build an application. The need for conventional programming is minimised, since the bulk of the interface management is absorbed in the underlying user interface model.

3 An Application Framework for Application Builders

We start with the definition of the concepts serving as the foundation for our UIMS. It is not an aim of this paper to propose the ultimate UIMS framework. Rather, we want to argue that the use of a framework allows the construction of an "open" application builder. Moreover, reflection makes it possible to migrate seamlessly from the user interface design level to the component design level.

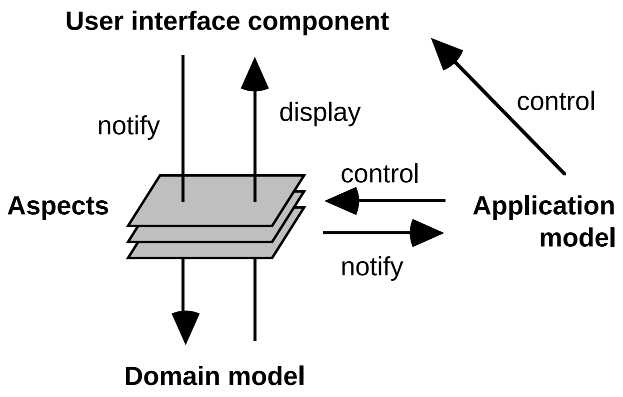


Figure 1: The Framework Architecture

For easy refinement of a framework, a set of orthogonal concepts is essential [JR91]. We adopt the terminology of VisualWorks[1], mainly because

[1]VisualWorks is a trademark of ParcPlace Systems Inc.

we prefer its clear distinction between the application model and domain model. We purify these concepts to fit them into the framework. In Fig. 1 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, aspects, is needed to separate the domain model from the user interface component. The relations between the four concepts are display, notify and control.

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 and network communication are included in the domain model as well. Several application models for one domain model can exist in parallel.

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 own: 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 [KP88, WGM88]. That is precisely the motivation for the relationships 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* relationship from the application model to the user interface component and the *display* relationship 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* 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* relationships (from the interface component to the domain model and from the aspect to the application model) and both of the *control* relationships (from the application model to the interface component and to the aspect).

4 An Example

In this section we will demonstrate all the concepts introduced above by using a simple example.

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 to increment or decrement the value. Fig. 2 shows how such an interface may look like.

To create this interface, the interface designer opens a new canvas (an

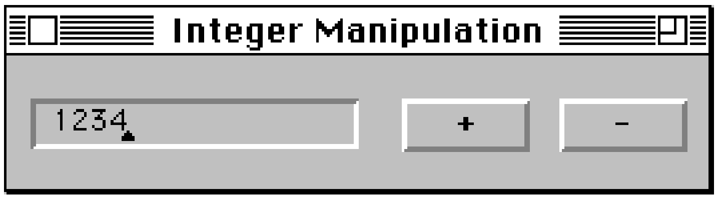


Figure 2: A Running Application to Manipulate an Integer

editor representation for the interface being developed) and draws it using the painter. The designer drags visual components (here two buttons and one text field) from a palette onto the canvas and positions them appropriately. Afterwards the interface desiner uses the properties editor to define properties of these components. Two major kinds of properties can be identified: visual properties and aspect properties. Visual properties influence the visual representation of a component, while aspect properties name the aspects (see Sect. 3). In this example the labels of the buttons and the font are visual properties, and the name of the aspect that refers to the integer value (held by the domain model) and the names of the aspects of the two buttons are aspect properties.

After concluding this procedure, the user has created the user interface for the application without writing a single line of code. 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 user interface 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 user interface 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. This 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).

The architecture of the user interface of the integer manipulation example can be expressed in our framework that was explained in the previous paragraph. Fig. 3 shows the interaction of the different components of this example.

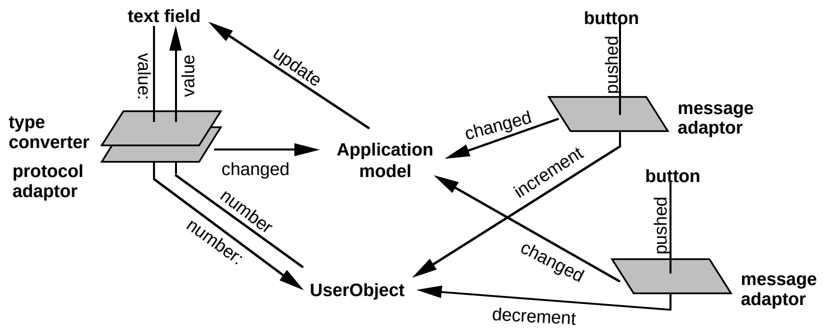


Figure 3: The Structure Behind the Number Application

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.

Aspects are needed to glue these user interface components together with the application and the domain model (see Fig. 3):

- A protocol adapter 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 adapter 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.

These `changed` messages eventually cause `update` messages being sent from the application model to the text field.

5 User-Defined Components

In the previous section we saw different examples of standard components. All standard components have common features: they can be dragged from a palette, they have a name, etc. The assembly of all these features defines the common interface the application builder framework expects of components. In other words, in order to play their role in the application builder, components have to conform to this common interface. More concretely the application builder requires the following information about a component:

- Name: for textual reference.
- Icon representation on component palettes: for visual reference.
- User interface: the interface of the component.
- Behaviour: the application model (see Sect. 3).
- Link with the domain model: specification of the domain model and the aspects that reference it.
- Properties editor: the meta application to set properties of the component.

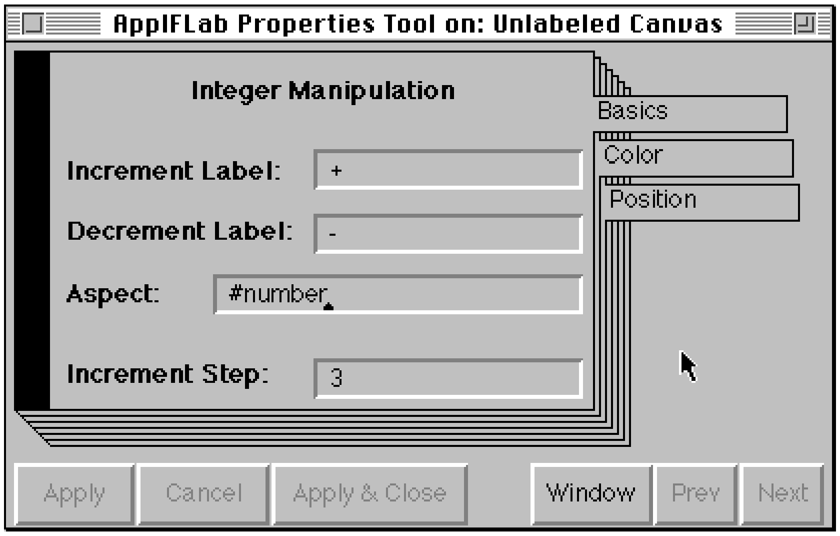
In regard to reflective object-oriented programming languages, this interface can be called the metaobject protocol [Kic92] for our application builder. We can identify different categories in this protocol. The first three items from the above list have to do with the visual representation and selection of the component. The last three items define the interaction of the component with the user and the domain model. In order to define this behaviour the component must have a link to the domain model. Furthermore, as we saw in the previous section, a component can be parameterised by different properties. These properties are filled in with the properties editor provided by the component.

The standard components of a general-purpose application builder almost never meet the requirements of the application designer, because they are too general in their semantics. As a consequence, using these components for the creation of domain-specific applications requires writing a lot of application logic. In absence of an application reuse mechanism, this domain-specific application logic is often duplicated in several applications, since not only the user interface, but also the application logic needs to be copied when an application is reused.

To solve the problem of code duplication, many application builders support the reuse of applications as subapplications. Although subapplications seem similar to (user-defined) components, they lack some of the important features of components. Subapplications are an important means to give names to groups of components. However, the philosophy of subapplications is to work as stand-alone entities. The contrary holds for components. Subapplications encapsulate their domain model and they do not allow the user to associate properties with the subapplication as a whole. Thus, applications can only be reused as-is. Changing the domain model of a subapplication requires the user to delve into the subapplication's implementation. Similarly, changing the visual representation of a subapplication can only be accomplished by changing the visual appearance of individual components in the subapplication's user interface. So, subapplications solve the problem of code duplication only partly, since application code needs to be written to adapt the reused application to specific needs. Therefore fully-fledged user-defined components are necessary.

We characterise a *user-defined component* as a subapplication which allows properties to be associated with it. These properties influence its visual representation, its behaviour and its link to the domain model. This means that a user defined component conforms to the protocol expected of components in general.

In general, self-made components are directed towards a specific problem domain. *Domain-specific components* are typically components that can be reused with little or no coding effort, since they encapsulate all the necessary component logic to live in a domain-specific application. To create a domain-specific application the user only needs to connect the domain-specific components to each other and to the domain model. Components are linked to each other by means of aspect properties.



Integer Manipulation

Increment Label: +

Decrement Label: -

Aspect: #number

Increment Step: 3

Figure 4: The Properties Editor of the Integer Manipulation Component

Suppose that we want to make an abstraction of the integer manipulation application of Sect. 4. The name of the component is "Integer Manipulation", the user interface is the one depicted in Fig. 2 and it has a behaviour as described in Sect. 4 (implemented as methods on a class).

The link with the domain model defines that the Integer Manipulation component expects to communicate with an aspect understanding the `value/value:` protocol. The properties editor is depicted in Fig. 4. With the properties editor the labels of the plus and minus buttons can be changed (both visual properties) and the aspect can be specified (aspect property). Also the number with which the integer value should be incremented or decremented each time one of the buttons is clicked can be provided.

In conclusion, there is a need for user-defined domain-specific components with their own behaviour and properties. Together, a set of user-defined components all targeted for the same domain can be grouped in order to define a *domain-specific application builder* (see Fig. 5).

Examples of such domain-specific application builders range from builders for the design of instrument controller applications [Joh94], database applications to simulation applications for children [CS95]. Especially in the area

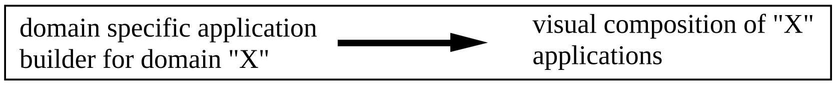

domain specific application builder for domain "X"
visual composition of "X" applications

Figure 5: Domain-specific Application Building

of domain-specific frameworks (i.e. object-oriented frameworks that target a particular problem domain) domain-specific application builders are becoming extremely popular [BW95]. They allow the visual instantiation of a framework into an application, which is considered as one of the hot topics in framework research.

6 Why is this Meta?

The problem with domain-specific application builders is that they are expensive to build. One way to solve this problem is to consider the construction of application builders as an application domain in itself. Not only will we need an object-oriented framework for application builders, we will also need the necessary components that together form an application builder for application builders. This latter obviously is called a *meta application builder* .

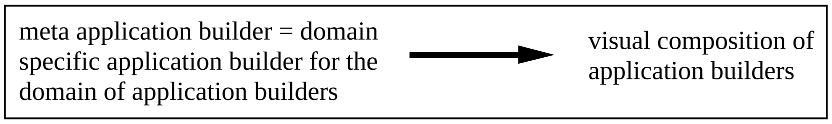

meta application builder = domain specific application builder for the domain of application builders
visual composition of application builders

Figure 6: Meta Application Building

What constitutes a meta application builder? Depending on the level of sophistication of the targeted application builder the meta application builder consists of several tools. Obvious tools can be identified such as a palette builder, i.e. an application that allows visual construction of component palettes, a component builder that allows grouping of components into a new component, aspect editor builder, a properties editor builder, or even a conventional code browser for those parts of the application builder that can not be defined visually. The following illustrates this using the proposed editor builder as an example.

Properties come mainly in two different kinds: visual properties and aspect properties. As we already said components are linked with each other and the underlying framework by setting aspect properties. So, a properties editor is essentially an application that sets, and gets, the properties of a component. Therefore, a properties editor is a metalevel application (see Fig. 7): whereas normal components manipulate aspects of some domain model, the components that make up a properties editor manipulate aspects of other components (i.e. properties of those components).

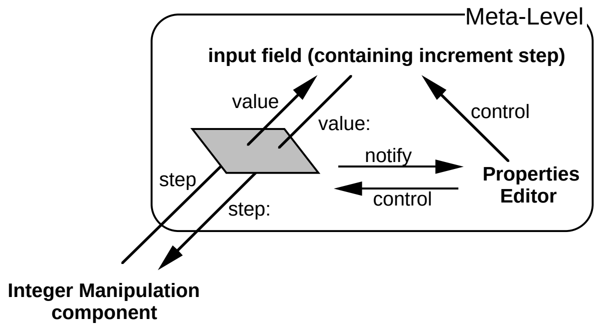

Meta-Level
input field (containing increment step)
value
value:
control
notify
step
Properties Editor
control
step:
Integer Manipulation component

Figure 7: The Properties Editor is a metalevel application

Typical properties editors are composed of fields that allow to set and get the identification of the edited component, fields for the visual properties, and fields for aspect properties. These fields are then also the components that can be found on the properties editor builder palette, i.e. the palette used for building properties editors (Fig. 8).

These editor components constitute a domain-specific application builder for building properties editors. Together the different editors (properties editor, palette editor, etc.) constitute the meta application builder which allows easy construction of user-defined components. If this meta application builder can, in its own terms, be extended with user-defined components, we can say it is a *reflective application builder* (see Fig. 9).

With a reflective application builder user-defined components can be

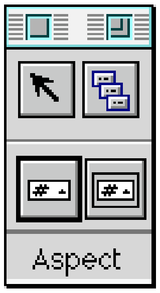


Figure 8: Properties Palette with Aspect Field Selected

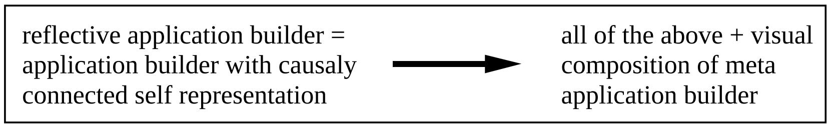

reflective application builder = application builder with causaly connected self representation
all of the above + visual composition of meta application builder

Figure 9: Reflective Application Building

made that can not only be used for easier construction of particular applications, but also for easier construction of application builders.

To illustrate the reflective use of the application builder, reconsider the properties editor of our Integer Manipulation component (see Fig. 4). Instead of using an input field for setting the increment step of the component, one could use the Integer Manipulation component itself. Since the input field and the Integer Manipulation component both use an aspect that refers to an integer number, they can be interchanged without affecting the behaviour of the properties editor. Only the user interface of the properties editor changes.

7 Implementation Issues

The implementation of our application development environment ApplFLab is an extension of VisualWorks/Smalltalk . The application framework introduced in Sect. 3 is adopted from VisualWorks and the screendumps shown in this paper were made while VisualWorks or ApplFLab was running.

Building user-defined components with VisualWorks is hard and requires a lot of manual coding. Expert programming skills are required, since adding user-defined components is not documented and VisualWorks does not provide any tools for component creation. The component designer has to specialise several classes of the application builder framework and has to maintain the implicit dependencies between these classes. In this regard, the VisualWorks application builder is a closed system. The metalevel interface is not made explicit and is, due to a lack of documentation, extremely hard to comprehend for somebody who is not a VisualWorks expert.

The current implementation of ApplFLab extends VisualWorks with a reflective application builder as described in the previous sections. The closed VisualWorks application builder was turned into an open implementation with reflective facilities, essentially by extending the application builder framework and adding the *component editor* , ApplFLab's metatool to create user-defined components. The component editor guides the component designer through all the steps that need to be taken for the construction of a user defined component. Essentially this leads to the provision of the information mentioned in Sect. 5: the component's name, user interface, behaviour, link with the domain model, properties editor and icon representation. The component editor incorporates a standard application builder to create the component's user interface and behaviour, and a reflective application builder to paint the component's properties editor (only painting, because all behaviour is absorbed by the builder).

The use of user-defined components has no significant effect on the performance of the applications that make use of them. ApplFlab's user-defined components are VisualWorks subapplications (so called subcanvasses) where the standard properties are replaced by user-defined properties. The user-defined properties are specified when the component's properties editor is painted with the component editor. No programming is required to associate properties with user-defined components. Although using VisualWorks subcanvasses to build user-defined components is an easy and a very natural way for the component designer, this approach has the drawback of being slow when setting up a user interface.

The use of components as models for other applications (see Fig. 7 and Fig. 1), gives rise to an interesting implementation issue. Since our implementation is an object-oriented program and components have two roles, i.e. being a subapplication and being a model for metalevel applications, multiple inheritance seems a very appropriate way to design components. VisualWorks uses another approach to make components play their two roles. It splits a component into two parts, the application model and the specification, each having specific behaviour corresponding to the role they play in the application builder. The application model embodies the behaviour of a component at runtime, while the specification encapsulates the behaviour of a component at creation and at building time. At painting time the user paints a component and sets its properties by means of the properties editor. All properties are stored in the component's specification. At building time the component's user interface is set up and its application model is created. The component's properties are copied from the specification to the application model. Conceptually, the application model and the specification share the properties, as would be the case with multiple inheritance.

Note that both a component's specification and its application model play their role at painting time. When a component is painted on a canvas, a running application model is created to represent it on the screen, and a specification is created to hold its properties. When the properties are changed through the properties editor, the application model is replaced by a newly created one (the application builder actually goes through a building phase) to reflect the changes. This is also very simmilar to the multiple inheritance approach, where the specification and the application model would be represented by only one object. ApplFLab's component editor generates both parts of a component and takes care of the copying of the properties from its specification to its application model.

8 Status and Future Work

ApplFLab has been used to create several domain-specific components for two specific problem domains: metro simulation and teleshopping.

Metro simulations can be set up by visual composition of components that represent metro stations and metro tracks. With the properties editors of these components the user links the tracks and stations to each other to form a metro network. A special run button can be triggered to start the simulation. The user then is able to watch metro trains riding through the network. The different stations and trains are displayed so that the user can monitor the ongoing activity.

Teleshopping applications are constructed with user-defined components such as shelves, shopping bags, cash registers, product information components and special buttons that put a product from a shelve into a shopping bag. The set of basic teleshopping components has been extended with (even more domain-specific) components to play CDs and videos. Some of these components have been reused in the properties editors of other teleshopping components to clearly illustrate the reflective nature of the ApplFLab application builder.

The above experiments are toy applications in nature. The following step is the use of ApplFLab in real world examples. We plan to launch a project in which ApplFLab is used to create user-defined components for a broadcast planning application.

The current version of ApplFLab provides all the required tools, but they are not sufficient for easy and fast component creation. After a series of experiments, it is clear that the component editor needs to be specialised towards component editors for repetitive tasks, such as the specialisation of standard action buttons to special-purpose action buttons, i.e. buttons that encapsulate behaviour towards a specific application domain.

Besides the specialisation of the component editor, we also believe that an adaptor editor is indispensable. ApplFlab's component editor provides a way to specify the link with the domain model through the use of aspects, but the set of aspects cannot be changed at this time.

9 Related Work

Application frameworks and user interface builders that make use of application frameworks are extensively described in various sources. Apple's MacApp [Sch86], Borland's OWL, ZINC, ET++ [WGM88], NextStep, VisualAge and VisualWorks are the most popular examples. All of these are general purpose application frameworks that can serve as the basis for a broad spectrum of applications.

Apart from these general-purpose application builders, there exist (ad hoc) application building tools for specific kinds of applications. Examples are Mathematica [Wol88] and KidSim [CS95]. These tools are neither extensible nor adaptable to different application domains as in most cases they are not explicitly linked to an extensible framework. Related to these are the various component-oriented software tools. The term component is used to describe any software entity that can be connected with other such entities with the aid of visual programming tools and/or a scripting language [CP95]. Examples include VisualBasic and Powerbuilder. These tools also suffer from their lack of extensibility and/or adaptability to a specific problem domain.

With the exception of the work described by Monroe, Karsai and Starringer [MG95, Kar95, Sta94] the creation of application building tools that are adaptable to some problem domain is still an open field. Starringer [Sta94] describes an application for calculating investment risks. The NextStep development environment was specialised such that it could be used by the end-user for customising the application. This way new calculations can be added without the help of expert programmers. Reconfigurable visual programming environments are discussed by Karsai [Kar95]. The research of reconfigurable style-specific programming environments is restricted to Monroe's work [MG95].

10 Conclusion

Reflective application builders are essential in the creation of domain-specific applications. After all, standard application builders cannot be customised towards a particular problem domain and current domain-specific programming environments cannot be tuned to other domains. Reflective application builders bridge the gap between the generality of the former and the domain-specificity of the latter. They support integration of different domain-specific application builders since they all share a common kernel. The integration of standard tools and domain-specific tools has the advantage that domain experts and software engineers work in the same medium. Each of them employs the same reflective user interface builder, but on a different abstraction level.

References

[BW95] Margaret M. Burnett and McIntyre David W. Visual Programming, Guest Editor's Introduction. *IEEE Computer*, 28(3):14–16, March 1995.

[CP95] S. Cotter and M. Potel. *Inside Taligent Technology*. Addison-Wesley, 1995.

[CS95] Allen Cypher and David C. Smith. KIDSIM: End User Programming of Simulations. In *Proceedings of the 1995 Conference on Human Factors in Computing Systems*, pages 27–34. ACM, ACM Press, 1995.

[Deu87] L. P. Deutsch. Levels of Reuse in the Smalltalk-80 Programming System. In Peter Freeman, editor, *Tutorial: Software Reusability*. IEEE Computer Society Press, 1987.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80, The Language*. Addison-Wesley Publishing Company, 1989.

[JF88] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object Oriented Programming*, 1(2):22–35, June 1988.

[Joh94] Gary W. Johnson. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*. McGraw-Hill Series on Visual Technology. McGraw-Hill, 1994.

[JR91] R. E. Johnson and V. F. Russo. Reusing Object-Oriented Design. Technical Report UIUCDCS 91-1696, University of Illinois, 1991.

[Kar95] G. Karsai. A Configurable Visual Programming Environment. *IEEE Computer*, 28(3):36–44, March 1995.

[Kic92] G. Kiczales. Towards a New Model of Abstraction in the Engineering of Software. In *Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architecture*, pages 1–11. Research Institute of Software Engineering (RISE), Information-Technology Promotion Agency, Japan (IPA), November 1992.

[KP88] Glenn. E. Krasner and Stephen. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, August 1988.

[LP91] W. R. Lalonde and J. R. Pugh. *Inside Smalltalk - Volume II*. Prentice Hall, 1991.

[LVC89] Mark. A. Linton, John. M. Vlissides, and Paul. R. Calder. Composing User Interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[Mey86] N. Meyrowitz. Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 21(11), pages 186–201. SIGPLAN, ACM Press, November 1986.

[MG95] R.T. Monroe and D. Garland. Style-Based Reuse for Software Architectures. Submitted to the 1996 International Conference on Software Reuse, April 1995.

[Rao91] R. Rao. Implementational Reflection in Silica. In P. America, editor, *Proceedings of the 5th European Conference on Object-Oriented Programming*, pages 251–267. Springer-Verlag, 1991.

[Sch86] K. J. Schmucker. *Object-Oriented Programming for the Macintosch.* Hayden Book Company, 1986.

[SDHDDM95] Patrick Steyaert, Koen De Hondt, Serge Demeyer, and Marleen De Molder. A Layered Approach to Dedicated Application Builders Based on Application Frameworks. In D. Patel, Y. Sun, and S. Patel, editors, *Proceedings of the 1994 International Conference on Object-Oriented Information Systems*, pages 252–265. Springer-Verlag, December 1995.

[Sta94] W. Staringer. Constructing Applications from Reusable Components. *IEEE Software*, 11(5):61–68, September 1994.

[Ude94] J. Udell. ComponentWare. *Byte*, 19(5):46–56, May 1994.

[WB90] Allen Wirfs-Brock. Panel: Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks. In Jerry L. Archibald and K.C. Burgess Yakemovic, editors, *Addendum to the Proceedings of the 5th Conference on Object-Oriented Programming Systems, Languages and Applications, and the 4th European Conference on Object-Oriented Programming*, pages 19–24. ACM/SIGPLAN, ACM Press, 1990.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++: an Object-oriented Application Framework in C++. In *Proceedings of the 3rd Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 23(11), pages 46–57. Association for Computing Machinary, ACM Press, November 1988.

[Wol88] S. Wolfram. *A System for Doing Mathematics by Computer.* Addison-Wesley, 1988.

Index