Experiences of Object-Oriented Development in C++ and InterViews

Brück, Dag M.

1989

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT–7418)/1-008/(1989)

Experiences of Object-Oriented Development in C++ and InterViews

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
March 1989

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
March 1989

Document Number
CODEN:LUTFD2/(TFRT–7418)/1–008/(1989)

Author(s)
Dag M. Brück

Supervisor

Sponsoring organisation
The National Swedish Board of Technical Development (STU project 87-2503)

Title and subtitle
Experiences of Object-Oriented Development in C++ and InterViews

Abstract

This paper describes our experiences with InterViews, an object-oriented package for implementing user interfaces written in C++. A comparison is made with PHIGS, a more conventional graphics standard. A strong interaction between base classes and derived classes is observed, notably base classes depending on the behaviour of the derived classes. The application is an interactive block diagram editor. It is used as a stand-alone graphical tool which generates equations for Simnon, a simulator for non-linear systems.

This paper has been submitted to OOPSLA'89 (Conference on Object-Oriented Programming Systems, Languages and Applications), October 2–6, 1989, New Orleans, Louisiana, U. S. A.

Key words
Object-oriented programming, User interfaces, C++, InterViews, Computer Aided Control Engineering

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
8

Recipient's notes

Security classification

Experiences of Object-Oriented Development in C++ and InterViews

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
Box 118, S-221 00 Lund, SWEDEN

E-mail: dag@control.lth.se

Abstract

This paper describes our experiences with InterViews, an object-oriented package for implementing user interfaces written in C++. A comparison is made with PHIGS, a more conventional graphics standard. A strong interaction between base classes and derived classes is observed, notably base classes depending on the behaviour of the derived classes. The application is an interactive block diagram editor. It is used as a stand-alone graphical tool which generates equations for Simnon, a simulator for non-linear systems.

1. Background

Developing real control systems is always a difficult task. Mathematical models and simulations are often used in the design and analysis of control systems. The use of computers for this purpose is called Computer Aided Control Engineering (CACE). The development of new CACE tools requires research on the basic concepts of modelling control systems, and the computer representation of control system models [Mattsson, 1988][Andersson, 1989]. Equally important is the choice of tools for developing these tools.

For research and prototyping, a combination of KEE and Common Lisp has proved effective. KEE provides a dynamic and interactive object-oriented development environment, including simple graphical output [IntelliCorp, 1986]. A practical engineering tool for designing control systems must be more economical than an expert system like KEE, so a leaner implementation is needed. C++ is a very good implementation language in this case because of its efficiency and support for object-oriented programming [Brück, 1987].

One of the remaining problem areas is the implementation of the user interface. The developer must choose among a few window managers and several graphics packages. In the on-going evaluation of different alternatives, this paper describes our experiences with InterViews [Linton and Calder, 1987], an object-oriented library for implementing user interfaces, written in C++ [Stroustrup, 1986] and running on the X Window System [Poutain, 1989].

The evaluation of InterViews was conducted by developing a block diagram editor for Simnon, a simulator for non-linear systems [Elmqvist et al., 1986]. Simnon is an interactive, command driven simulation package with its roots in the 1970's; Simnon is still very much state-of-the-art for continuous simulation, but has no graphical input. The block diagram editor is not integrated with Simnon, and therefore reasonably sized for evaluation purposes. This paper also contains a comparison with an earlier evaluation of PHIGS (Programmer's

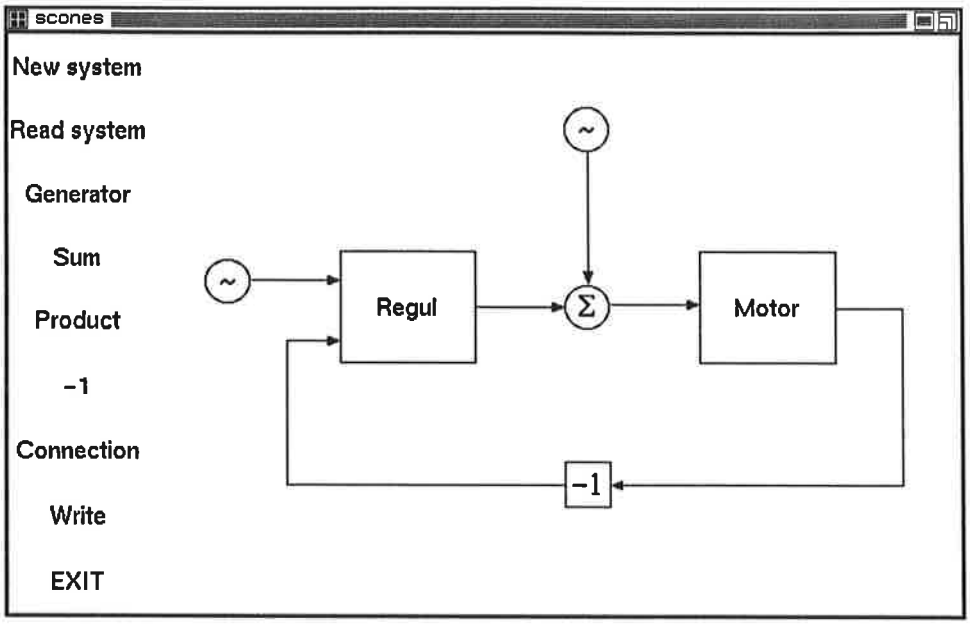

Figure 1. Screen dump of a simple block diagram.

Hierarchical Interactive Graphics Standard) in a similar application [Brück, 1988]. A parallel effort aims at developing a block diagram editor for Simnon on the Macintosh, using Object-Pascal and MacApp. Previous work has also explored continuous panning, scrolling and zooming of block diagrams on a high-performance workstation. The concept of information zooming was introduced, meaning that the information contents of a block changes depending on its size on the screen [Elmqvist and Mattsson, 1989].

2. The application

A key concept in Simnon is the *system*, which corresponds to a mathematical model of the reality being studied. A system is described in a special modelling language. There are continuous systems based on differential equations and discrete systems based on difference equations. A third type is the connecting system, which is used to form compound systems from enclosed continuous or discrete systems. Every Simnon system is stored as a separate text file.

The connecting system is often visualized (with pencil and paper) by drawing a block diagram. Unfortunately, the drawing must still be transformed into statements of the modelling language. The block diagram editor can produce simple forms of Simnon's CONnEcting System, hence the name *Scones*.

Figure 1 shows a simple block diagram in Scones. There is a fix command menu on the left side, and a drawing area for the block diagram on the right. Systems are represented by large annotated boxes. Special symbols represent the sum (Σ), product (Π, not shown in Figure 1) or negation (-1) of signals. General expressions are represented by generator symbols ($\sim$).

When creating a block diagram, the user can either create a new system in which case Scones will make a template file, or read an existing file in which case Scones will extract properties necessary for drawing the block diagram. Scones knows the name of the system, and maintains for each system a list of terminals (inputs and outputs) which can be connected to terminals of other systems. The connections define the interaction between the systems enclosed by the connecting system. A sequence of connected special symbols

```
CONTINUOUS SYSTEM Regul
"Filename pid.t
"Created Fri Feb 10 14:14:51 1989
INPUT y_ref y
OUTPUT u
END
```

Listing 1. Simnon code for the regulator system. Comment lines begin with a double quote (").

```
CONNECTING SYSTEM Consys
"Filename consys.t
"Created Fri Feb 10 14:30:33 1989
TIME t
"System: Regul
y_ref[Regul] = if t > 0 then 1 else 0
y[Regul] = -y[Motor]
"System: Motor
u[Motor] = u[Regul] + sin(t)
"Generator: if t > 0 then 1 else 0
"Generator: sin(t)
END
```

Listing 2. Simnon code for the connecting system.

are transformed into an arithmetic expression in the connecting system. It should be noted that Scones completely ignores the equations that define the behaviour of a continuous or discrete system. Scones also defines a global time variable t in every connecting system.

The block diagram in Figure 1 represents a servo constructed from a motor and a regulator. A generator provides a step in the regulator's reference value y_{ref}. The control signal from the regulator y is influenced by a load disturbance from another generator. The measured value from the motor u is negated. The template system for the regulator (without equations) is shown in Listing 1. The connecting system produced by Scones is shown in Listing 2. The template code for the motor is very similar to the code for the servo, and therefore not shown.

3. InterViews and PHIGS

InterViews is an object-oriented user interface package [Linton and Calder, 1987]. It provides the basic building blocks for implementing a wide variety of user interfaces. Basic objects derived from the base class *Interactor* can display a graphical image and accept input events. Composite objects derived from class *Scene* can display a complex image by combining other objects (including scenes).

Scenes defined in InterViews can arrange interactors in many ways: side-by-side horizontally (an HBox) or vertically (a VBox), one stacked above the other (a Deck), or framing an interactor (a Frame). Every interactor has a predefined natural size, but may stretch or shrink within specified limits. This means that a scene can adapt to available space by stretching or shrinking its components. Glue objects can be inserted to improve the layout. Other "high-level" user interface objects are scrollers and panners that change the view of a scene, different types of buttons, pop-up menus, and a string editor.

Comparing InterViews with an established graphics standard such as PHIGS [Brown,

1985] is like comparing apples and oranges; the comparison is interesting though, as either InterViews or PHIGS may be the best alternative in a particular application. Superficially, the similarities are striking: both InterViews and PHIGS provide

- Hierarchical structure of graphics. Complex images are constructed by combining simpler objects.
- Reuse of a graphical object in different contexts, and multiple views of a single object.
- Event mode input.

The main difference is in the degree of "object-orientedness." InterViews is fully object-oriented, whereas PHIGS can be classified as object-based [Wegner, 1987]. Graphical objects in PHIGS (called structures) are manipulated by a fixed set of operations, contain only graphical information, and their storage is managed by the PHIGS runtime system. With InterViews, classes derived from class Interactor add behaviour to graphical objects, and can directly represent the real-world object; no separate graphical object hierarchy is needed.

Interactor objects in InterViews are more "live" than structures in PHIGS. When a graphical object changes, it sends a *Change* message to its parent (enclosing scene). InterViews will then send *Redraw* messages to all affected interactors, including the one that was changed; the interactors draw images that reflect their internal state. Redraw messages are also sent on demand from the window manager, for example, when hidden interactors become visible. With PHIGS, the application program must edit the contents of separate structures. The PHIGS system will generate the image by traversing its internal data structures, either on command from the application program, or "when necessary." It is probably easier to use specialized graphics processors or to distribute processing to intelligent graphics terminals in PHIGS, than it is in InterViews.

Similarly, input events are sent directly to the target interactor in InterViews. In PHIGS, the application program will get the identifier of the target structure and of all ancestor structures of the target. The application program is responsible for identifying related objects in its own world. InterViews also contains a set of PHIGS-like graphical objects, derived from class *Graphic*. Apparently, class Graphic does not handle input events, so interactors were used in this project.

Another important difference is the positioning of objects. PHIGS objects are positioned at (x, y); multiple local coordinate systems may be used. In InterViews, objects are typically positioned relative another object, without bothering about the exact coordinates; the object may in fact move around or be reshaped as available space increases or decreases. The InterViews approach is normally much more convenient, and interacts better with the window manager. The strengths of PHIGS are its powerful 3D capabilities, and its handling of different projections. Good PHIGS implementations are also significantly more efficient in drawing complex images. Filters are used in PHIGS to control what objects should be visible, pickable, or highlighted. Filters are hardly needed in InterViews, as the graphical image is generated by user written routines that easily adapt to the properties of the corresponding objects. In PHIGS, filters are quite useful.

In short, PHIGS can be regarded as a powerful standard for drawing graphics, and InterViews as a powerful tool for building user interfaces.

4. System design

Scones was designed with simplicity and ease of implementation in mind. It has few features and the user interface is simple. Interaction is mouse based, except for input of text strings. The use of Scones is strongly influenced by the way you draw block diagrams manually, the modelling concepts in Simnon, and the user interfaces of other drawing programs. Internal operation is event driven, using the default event dispatcher of InterViews. Scones was

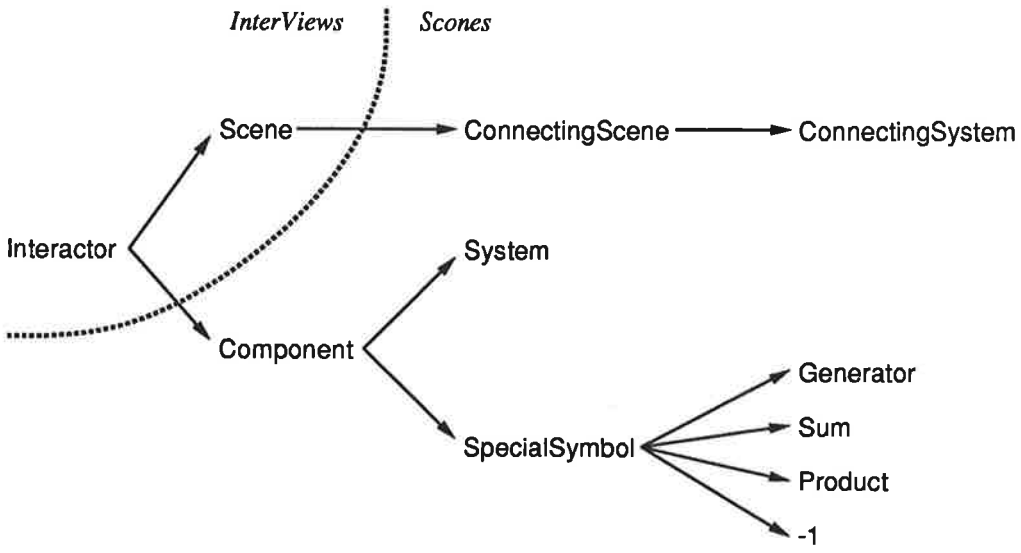

Figure 2. The class hierarchy in Scones.

implemented entirely with InterViews and there are no direct calls to the underlying X Window System.

An important objective was closeness to InterViews. Most classes used for representing the block diagram were designed as step-wise augmentations of predefined InterViews classes. Mixing attributes related to the real-world objects being modelled and the graphical attributes is appropriate in this application; in other applications it may be desirable to separate the graphical aspects, for example, to take advantage of distributed graphics processors. The availability of multiple inheritance would probably lead to a design with looser coupling between graphical and modelling aspects. It would then be possible to build a class hierarchy based on the modelling aspects, inheriting graphical aspects as needed from InterViews.

The major class hierarchy in Scones is shown in Figure 2. *Interactor* is the base class of all graphical objects. Class *Component* represents the common behaviour of all objects in a block diagram. Typical attributes are terminals (the endpoints of a connection), operations on all terminals of a component, and handling of events. Component is an abstract base class (no objects can be directly instantiated) and most operations are realized in derived classes, for example, to generate the equations of the connecting system by following connections. Class *System* represents a continuous or discrete system in Simnon. One specialization is the ability to fire up the editor on the corresponding Simnon text file. The main purpose of all other components is to tie together connections. These attributes are represented by class *SpecialSymbol*, but geometrical shape and arithmetic meaning are realized by derived classes. Class *Generator* has more features than other special symbols (e.g., it can be edited), and should probably have been derived directly from class Component.

A scene in InterViews is essentially an arranger of other objects; this definition also applies to the connecting system of Simnon. The properties most closely related to the operation of InterViews are collected in class *ConnectingScene*. Additional properties related to connections and the generation of equations were collected in class *ConnectingSystem*. The division into two levels of derivation was motivated by the problems in realizing all the needed behaviour of an InterViews scene. Connections are not regarded as objects like systems or summation symbols (and are therefore not interactors), but rather as an attribute of the connecting system. This distinction is probably wrong; many operations (e.g., deletion) would be easier to implement if connections were represented by interactors. The user interface could also be improved if connections responded to mouse clicks, etc.

5. Experiences

The first question that arises when you start using a new software package is "What can I do?" The second is "How should I do it?" Graphics with InterViews can be realized in three complementary ways:

1. InterViews provides a rich set of ready-to-use building blocks, for example, text messages, buttons, and a string editor. These standard interactors are easy to use, easy to integrate (e.g., to create an input form), and behave as expected.

2. Simple user defined graphical objects are derived from class Interactor. A few low-level methods must be implemented, such as, Redraw. Certain attributes of the interactor must be initialized by the user, for example, the shape object and interest in input events.

The methods and attributes of the low-level objects are not difficult to understand separately, but their use should be better documented to the benefit of new users. When no output at all is produced, it may not be obvious that the real cause was forgetting to initialize the shape member variable. Misuse of attributes, failure to implement a method, or performing initializations in the wrong method, may initially pass unnoticed; in some other context, tried and "debugged" classes may fail for some unexpected reason.

3. Composite graphical objects that contain other interactors are called scenes. InterViews provides many useful types of scenes, but apparently not a scene that simply puts an interactor at position (x, y) which was needed in Scones.

Implementing a scene is considerably more complex than just drawing some graphics. Firstly, the scene must manage a collection of inserted interactors. A number of operations may require interaction with the enclosed objects, for example, shape calculations. Secondly, the derived scene interacts intimately with its base class and the low-level routines of InterViews. The user written scene must provide a number of services for insertion, deletion, changes, reconfigurations, reshaping, etc. Furthermore, the scene must have a fairly complete set of operations to be operable at all; few shortcuts are possible. On the other hand, once done it is quite easy to comprehend, and not too difficult to redo for a different application.

Object-oriented programming is apparently more complicated than normally presented, i.e., as simply inheriting behaviour from the base class, or as the base class providing a template for the interface of derived classes. This application shows a strong coupling between base class and derived class; in particular, the function of the base class relies on a properly implemented derived class. This is exactly why the keyword `protected` was introduced into C++; to distinguish class members that must be accessed by derived classes, but not by code outside these classes. The introduction of multiple inheritance in C++ will hopefully enable a design with less coupling.

The problems with strong coupling are common in any application where code is reused, and obviously not typical for object-oriented programming. The need for high-quality design and documentation of generally used base classes is pronounced. Object-oriented programming does make it easier to reuse existing code but the designer of a useful base class must anticipate future needs, for example, by declaring methods `virtual` in C++. One may say that object-oriented programming will give you less trouble with the past, and more trouble with the future.

InterViews is a well-designed package, and most problems are due to lack of documentation (about the average UNIX standard). A major improvement would be a description of the internal operations of InterViews, e.g., in the form of a data flow graph. This would give more insight in the interaction between objects, and the intended use of certain methods — what happens when a window is resized? Currently, a major source of documentation is the InterViews code. There are a number of overview papers related to InterViews [Linton

and Calder, 1987][Vlissides and Linton, 1988][Linton et al., 1989], and a lively mailing list on Internet.

Little effort was needed to learn how to use InterViews and implement an acceptable user interface, compared to our previous experiences with PHIGS in a similar application. The new user interface is also much improved. Object-oriented programming is well suited to implementing user interfaces, and this application is close to the basic concepts in InterViews. Numerous revisions of the program has shown that it is easy to extend the user interface and to add new graphical objects. A considerable amount of time was spent on restructuring existing classes. Two features of InterViews have not yet been evaluated: perspectives for changing the view of a graphical object, and persistent graphics for saving graphical objects on a file.

A reasonable block diagram editor has been implemented in three months, including time to learn InterViews. Scones contains 987 lines of header files (mostly class declarations) and 2347 lines of other code. Users find the program somewhat slow, but it is unclear whether this is because of deficiencies in InterViews or in the X server.

6. Conclusions

InterViews is a powerful object-oriented package for implementing user interfaces. It provides a set of ready-to-use building blocks (e.g., text messages, buttons, a string editor), and simple graphical objects are relatively straight-forward to implement. Non-standard composite graphical objects are considerably more difficult, mainly because of missing documentation.

PHIGS is more efficient and has powerful 3D primitives, but PHIGS is not tailored at implementing user interfaces. Comparing two similar applications, InterViews is easier to use and yields a better user interface.

A surprising experience was the strong interaction between base classes defined in InterViews and derived classes defined in the application. The derived classes not only inherit behaviour, they must also provide services to the base classes and the InterViews system. This coupling stresses the need for good documentation, in particular documentation aimed at the class developer.

Acknowledgements

I am grateful for comments on the manuscript by Sven Erik Mattsson, Mats Andersson, Andrew Koenig and Mark Linton. This work was supported by the Swedish National Board for Technical Development (STU).

References

ANDERSSON, MATS (1989): "An Object-Oriented Modelling Environment," *Proc. 1989 European Simulation Multiconference*, June 7–9 1989, Rome, Italy.

BROWN, MAXINE D. (1985): *Understanding PHIGS*, Template Graphics, San Diego, CA, USA.

BRÜCK, DAG M. (1987): "Implementation Languages for CACE Software," CODEN: LUTFD2/TFRT-3195, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, DAG M. (1988): "Modelling of Control Systems with C++ and PHIGS," *Proc. USENIX C++ Conference*, October 17–20 1988, Denver, CO, USA.

ELMQVIST, HILDING, KARL JOHAN ÅSTRÖM and TOMAS SCHÖNTHAL (1986): *Simnon User's Guide for MS-DOS Computers*, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, HILDING and SVEN ERIK MATTSSON (1989): "Simulator for Dynamical Systems Using Graphics and Equations for Modeling," *IEEE Control Systems Magazine*, **9**, 1, January 1989.

INTELLICORP (1986): *KEE Software Development System User's Manual*, IntelliCorp, Mountain View, CA, USA.

LINTON, MARK A. and PAUL R. CALDER (1987): "The Design and Implementation of InterViews," *Proc. USENIX C++ Workshop*, November 9–10 1987, Santa Fe, NM, USA.

LINTON, MARK A., JOHN M. VLISSIDES and PAUL R. CALDER (1989): "Composing User Interfaces with InterViews," *IEEE Computer*, **22**, 2, February 1989.

MATTSSON, SVEN ERIK (1988): "On Model Structuring Concepts," *Proc. 4th IFAC Symposium on Computer-Aided Design in Control Systems*, August 23–25 1988, Beijing, P. R. China.

POUTAIN, DICK (1989): "The X Window System," *BYTE*, January 1989, 353–360.

STROUSTRUP, BJARNE (1986): *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, MA, USA.

VLISSIDES, JOHN M. and MARK A. LINTON (1988): "Applying Object-Oriented Design to Structured Graphics," *Proc. USENIX C++ Conference*, October 17–20 1988, Denver, CO, USA.

WEGNER, PETER (1987): "Dimensions of Object-Based Language Design," *Proc. OOPSLA'87*, October 4–8 1987, Orlando, FL, USA.