Revised version in Proceedings of the 1993 NZCS Conference, August, 1993, pp. 465-478.

# Integrated object-oriented software development in SPE

**John C. Grundy and John G. Hosking**
Department of Computer Science
University of Auckland
Private Bag 92019, Auckland
jgru1@cs.auckland.ac.nz

---

ABSTRACT
SPE is a software development environment which supports multiple textual and graphical views of a program. Views are kept consistent with one another using a mechanism of update records. SPE is useful throughout all phases of the software development life-cycle. It provides support for conceptual level object-oriented analysis and design using diagrams, visual and textual programming, hypertext-based browsing, and visual debugging, together with a modification history. SPE is implemented as a specialisation of an object-oriented framework and provides an environment for Snart, an object-oriented programming language.

---

## 1. Introduction

CASE diagrams are useful in all phases of the software life-cycle. In object-oriented (OO) programming, for example, diagrams illustrating class inheritance, association and aggregation relationships are an invaluable aid in understanding program structure [Coad and Yourdon 1991; Wilson 1990]. A natural extension of using diagrams to explain programs is to use *visual programming* as a means of constructing systems. Example visual programming systems include Fabrik [Ingalls et al 1988], Prograph [Cox et al 1990], and Garden [Reiss 1986]. OO structure diagrams are thus useful for analysing and designing systems [Henderson-Sellers and Edwards 1990; Wasserman et al 1990], visually building OO structures [Grundy and Hosking 1991 and 1992], browsing OO programs [Fischer 1987], visualising and debugging OO systems [Haarslev and Möller 1990; Ingalls et al 1988], and documenting and explaining systems at a high level of abstraction [Booch 1991; Wilson 1990]. Textual programming is also useful for describing the detailed aspects of software, such as implementing program functionality and detailed software documentation.

Much current research in software development environments centres on integrating these graphical and textual approaches to defining software [Ratcliffe et al 1992; Grundy and Hosking 1992]. This allows software developers to use the most appropriate representation of a program at during different development phases. Different representations must be kept consistent by the environment, however, so developers do not try and use or modify inconsistent views of software. A common to achieving this view integration is by using structure-oriented (or syntax-directed) editing for both kinds of program representations [Ratcliffe et al 1992; Reps and Teitelbaum 1984; Magnusson et al 1990]. Structure-oriented editing has not been widely accepted by programmers, however, as editing is quite restrictive [Welsh et al 1991; Arefi et al 1990; Minör 1990]. We adopt a less restrictive approach which

uses interactively-editing graphical views and free-edited and parsed textual views of programs. Both representations are kept consistent using a novel update record mechanism which propagates changes between all views of a program sharing common information.

In this paper we describe SPE (Snart Programming Environment) which supports multiple textual and graphical views of an OO program, integrated so that information in each view is consistent with that in other views. SPE provides a software development environment in which high-level OO program structures are analysed, designed, constructed and browsed using graphical class diagrams. These class diagrams show some of the classes that make up an OO program, features of these classes (i.e. attributes (data) and methods (procedures) of the class), and relationships between classes. Code-level class interfaces (defining a class's generalisation classes and features) and method implementations (procedural code) are programmed with free-edited text. Full consistency management between different phases of development is provided with design changes automatically modifying class and method interfaces and vice-versa. SPE also includes support for static program visualisation and browsing, dynamic program visualisation for debugging, and change documentation including a persistent history of program modifications, all integrated within one environment.

In the next section we describe the different types of program view available. This is followed by an example analysis, design and implementation of a simple drawing program using SPE. A description of view construction mechanisms, view navigation, and program complexity management and browsing is presented. Section 6 describes SPE's novel consistency management system and its uses. Brief descriptions of run-time support facilities and the architecture underlying SPE are followed by conclusions and discussion of current and future research.

## 2. Views in SPE

SPE allows a programmer to construct multiple textual and graphical views of a program. Fig. 1. is a screen dump showing two graphical and three textual views of a program implementing a simple drawing package. Each view renders a subset of the total program information. The *window-root class* view shows some of the inheritance and aggregation relationships between various classes of the drawing program. Information in a view may overlap with information in other views. For example, the figure and drawing window classes appear in both graphical views. An underlying *base view* integrates all the information from each view, defining the program as a whole. If shared information is modified in one of the views, the effects of the modification are propagated, via the base view, to all other interested views to maintain consistency. This consistency mechanism is described in Section 6.

Fig. 1. also illustrates some of the components that can make up a graphical view. Icons represent classes with abstract classes (e.g. `figure`) being distinguished from concrete classes (e.g. `window`) by a grey icon border. Class icons can contain names of methods and attributes (features) of the class, including those defined by the class or those inherited. Generalisation relationships are represented by bold arrows (e.g. between `drawing_window` and `window`). A variety of client-supplier relationships can be represented. These include aggregates (for example, the `figures` feature of `drawing_window` holds a list of `figures`); feature calls between classes (i.e. one class calling a method or fetching an attribute belonging to another

class) (for example, the `draw` method of `drawing_window` calls the `add_pic` method of `drawing_window`); and abstract class associations (`drawing_window` makes use of `button`) that can be used to represent arbitrary class relationships.

Also shown in Fig. 1 are a class definition, a method implementation, and a documentation textual view, the latter being arbitrary text associated with a program component. The class interface textual view defines class generalisation classes (parents) and class attributes and methods. The method implementation view defines the procedural code implementing a class method. The documentation view supplies arbitrary text which describes the purpose of classes and/or class features. The text view syntax used is that of Snart, an object-oriented extension to Prolog with a C++ like syntax [Grundy and Hosking 1992]. These different textual forms[1] can be mixed in one view or, as in Fig. 1., placed individually in separate views.

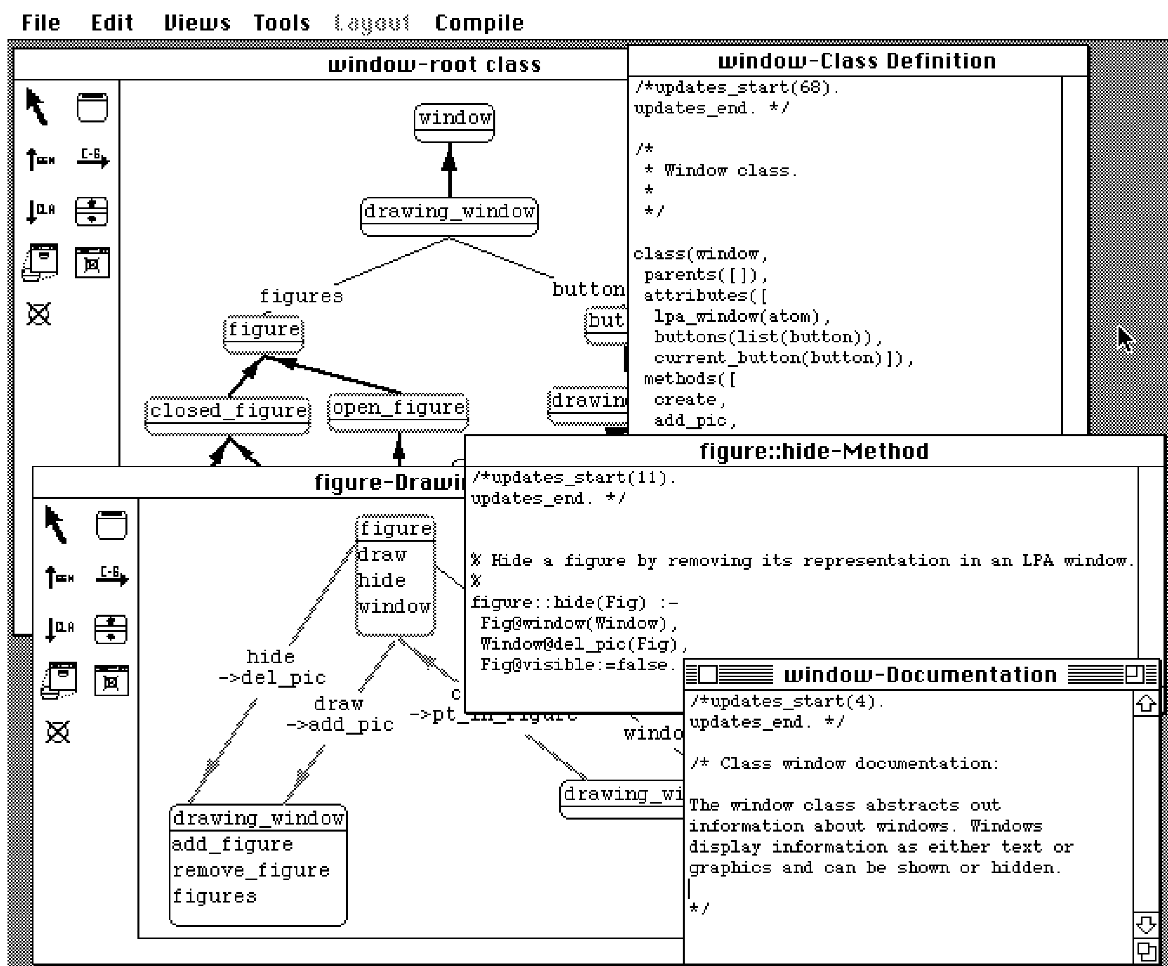


Fig. 1. An example SPE graphical and textual program views.

[1]A text form is some text describing one aspect of a program component. A class can have documentation and code text forms, a method can have code and documentation forms and other elements can have only documentation forms.

## 3. Integrated Analysis, Design, Implementation and Maintenance in SPE

The ability to generate an arbitrary number of views combined with the wide range of representations for viewing program components makes SPE useful throughout the program development cycle. Graphical views, such as the *window-root class* view, can be used during problem analysis to map out classes and their high level relationships, with documentation views providing more analysis detail. During design the analysis diagrams can be refined, either in the existing views, or in new design-oriented views such as the *figure-draw* view. Class definition and method implementation views allow detailed code to be added during late design and implementation.

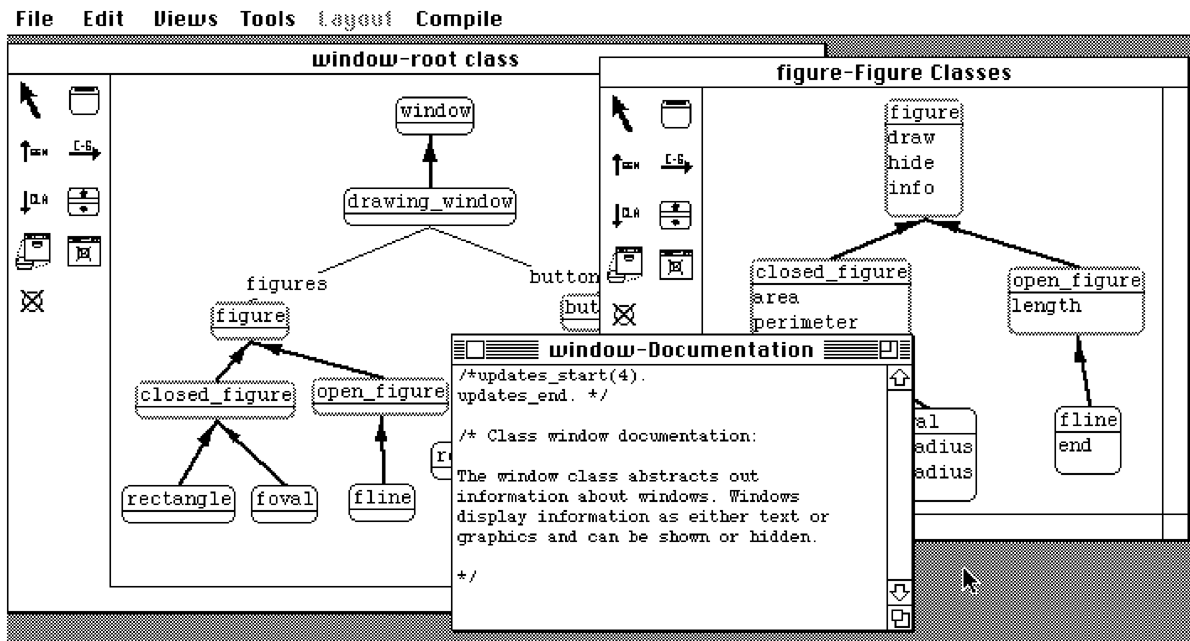


Fig. 2. Examples of the drawing program class hierarchy and extended analysis views.

For example, the first step in an analysis of the drawing program is to determine the class hierarchies required. A special `drawing_window` class is derived from the window class, and the `figure` and `button` hierarchies are defined. Figures can be specialised to `open_figure` and `closed_figure` figures, and buttons to `drawing_button`. Fig. 2. shows this class hierarchy with the `figures` and `buttons` attributes of `drawing_window`. The next step is to extend this hierarchy (possibly using extra views) to include the major relationships between classes and important class attributes and methods. At the analysis stage documentation about the purpose of classes and relationships can be added using textual views for a particular class, as shown in fig. 2.

After performing an analysis of the drawing program we can proceed to specify a design for its implementation. Extra detail is added to the various associations between classes, for example, the names to refer to them by and how they are to be implemented (as attributes, local or argument references, or by a feature call). Extra features and relationships between classes are introduced to implement various tasks. The analysis-level diagrams and documentation can be retained or new views created by copying information and extending it. The documentation added at the analysis phase can also be extended to describe more detailed program structure. Fig. 3. shows extended design-level views, some constructed by

copying the analysis-level diagrams. The window-Design Relationships view has extended the features of the window, figure and button classes to incorporate design-level functionality (such as picture manipulation in windows) and data (references from figures and buttons to their owning window). The figure-Drawing view describes feature class between classes to illustrate the functionality of some drawing program classes graphically.

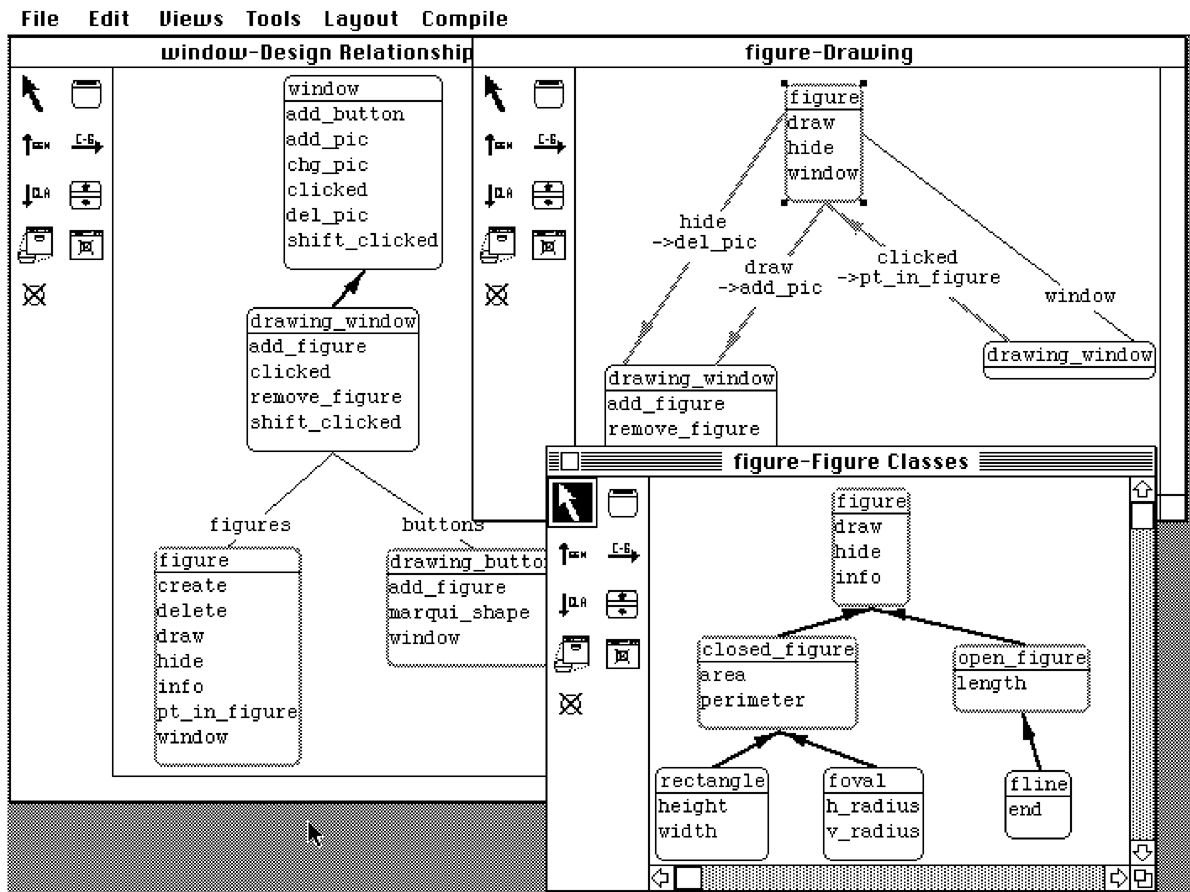


Fig. 3. Examples of design-level diagrams.

To implement a design in Snart the design diagrams can be extended to describe the actual types of client-supplier connections between classes. Class definition textual views are added to describe the complete class interface. Textual views are created in a similar manner to graphical views and consist of one or more text forms for different program components. Fig. 4. shows a class definition and methods implemented for the drawing program. Note that the figure::hide-Method view also shows the window::del_pic method which figure::hide calls. Textual views allow programmers to represent code in more than one view to facilitate viewing and understanding of programs.

When maintaining Snart programs in SPE, changes can be made at the analysis, design or implementation levels. SPE propagates analysis-level changes to the design, design changes to implementation views and implementation changes to design views. SPE also records changes against program components and provides browsing facilities for reviewing the modification history of a program. Section 6 discusses this consistency management system and program history in more detail.

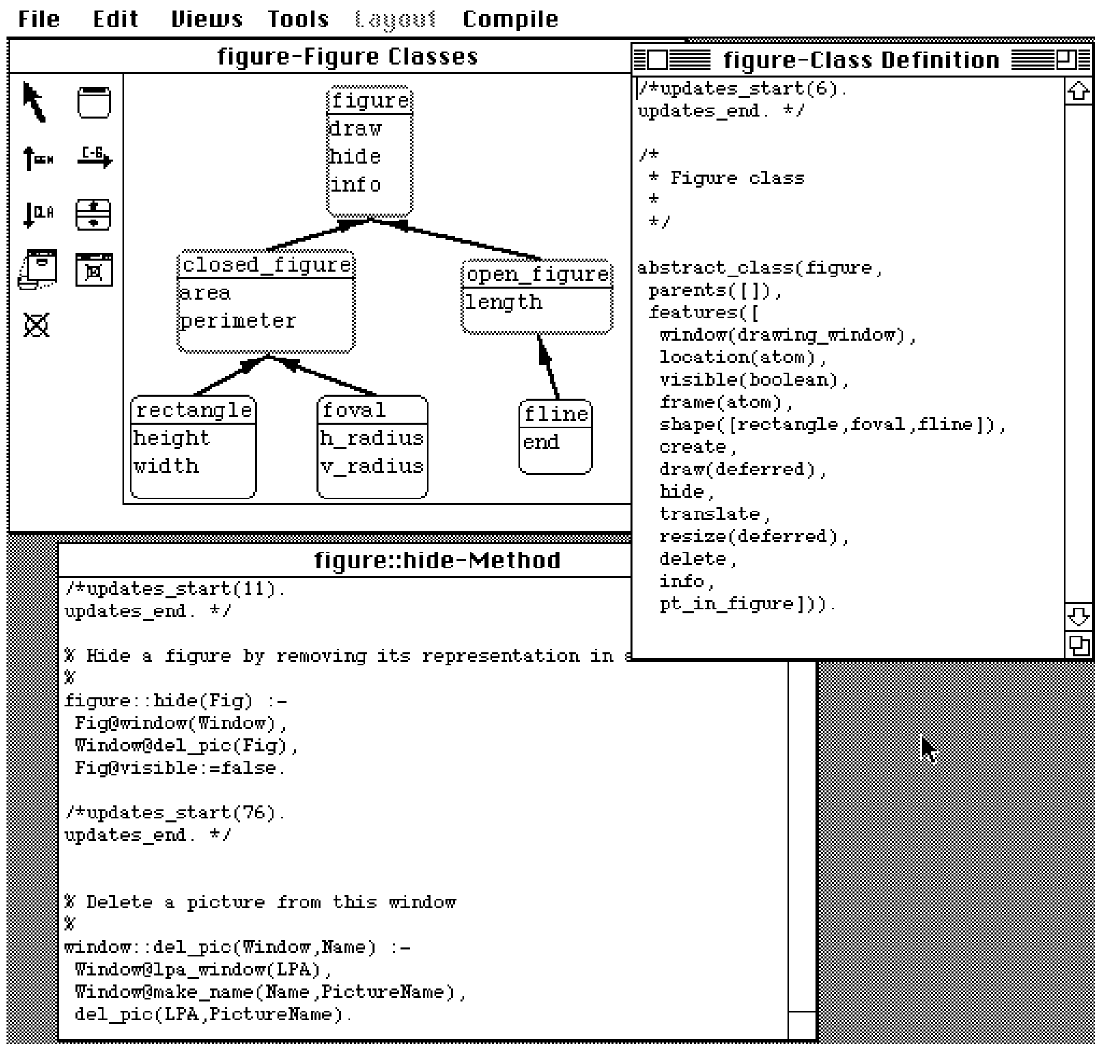


Fig. 4. Class definition and method implementation views for the drawing program.

Construction of specialised graphical or textual views focusing on particular aspects of the program can assist programmers in understanding the program and repairing faults. The close integration of textual and graphical forms allows programmers to work in whichever representation they feel most comfortable, for example designing using text, with graphical views for documentation, or vice-versa. The usefulness of this approach is dependent on the ease in which views can be constructed and accessed, and their information integrated in a consistent manner with that of other views. These capabilities are described in the following sections.

## 4. View Construction and Visual Programming

The user of SPE may construct any number of views of a program. SPE programs are incrementally saved to and loaded from database storage. Thus only a subset of a total program and its views need be in memory at any one time. Each view may be hidden or displayed using the navigation tools described in the next section. Elements may be added to a view using a variety of tools, with a full undo-redo facility allowing any modification to be

undone. SPE supports two types of view-element addition: the extension of a view to incorporate program elements already defined in another view (i.e. browser construction and program visualisation); and the addition of new program elements (i.e. visual programming).

Each graphical view has a palette of "drawing" tools which are primarily used for the addition of new program elements. Classes and generalisation relationships are added to views interactively and classes can be selected and dragged in a similar manner to figures in a drawing package. Client-supplier relationships are added between classes interactively. A dialogue is used to specify information about the relationship including its name (if any), its arity (the number of classes participating in the relationship), whether it is inherited from an ancestor class, and the type (call, aggregate, abstract). Feature names may be added to class icons and can include features the class inherits (i.e. from its entire interface). Tools are also provided for creation of new views and removing elements from a view or completely from the program.

Class icons have a click region at their base (discussed in the following section) which may be used in conjunction with the selection tool to add to a view information related to that class which has been defined in other views. This includes feature names, generalisations, specialisations, and the various forms of client server relationships. Thus specialised views of an existing or partially developed program can be rapidly constructed focusing on one or more aspects of that program. The view may then be used to extend or modify the program using the visual programming tools. The *figure-draw* view in Fig. 1., for example, is a specialised view showing the interaction of methods in each of the `figure` and `drawing_window` classes. Programmers lay out views themselves to obtain the most useful visualisation of programs and SPE automatically lays out expanded information (with programmers able to move these expanded objects to suit).

Textual views are created in a similar manner to graphical views, but consist of one or more text forms, rather than icons and "glue". Textual views are manipulated by typing text in a normal manner, i.e. a "free-edit" mode of operation in contrast to the "structure-edit" approach of the graphical tools[2] and then parsed to update base program information. This contrasts to most other environments, such as Dora [Ratcliffe et al 1992], Mjølner [Magnusson et al 1990], and PECAN [Reiss 1985] which use structure-editors for both graphics and text. Methods are implemented in the same way as class definitions and can be added to the same textual view as a class definition or have their own view (and window). Class definition and method textual views may be typed in from scratch or may be generated automatically from information about the class entered in other views and then extended in a free-edit fashion.

## 5. View Navigation and Browsing

As SPE allows an arbitrary number of views to be created for any class or feature, programs can become complex. Programmers must be able to locate information easily and be able to gain a high-level overview of different program aspects. SPE's approach is to use the program views themselves in a hypertext-like fashion as the basis for browsing.

---

[2]They also have an alternative, high-level structure-oriented style of editing using menus for manipulating individual text forms.

Class icons in graphical views have “click-points” which cause some pre-determined action carried out when clicked on (similar to Prograph’s dataflow entities [Cox et al 1990]). Fig. 5. shows the different click-points for a representative class icon.

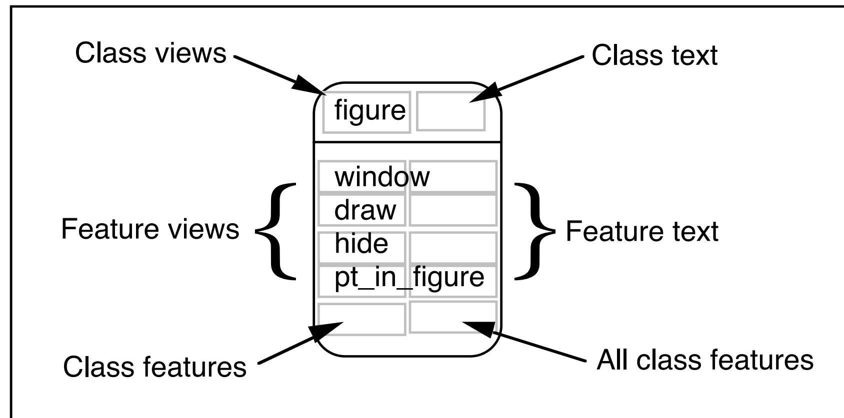


Fig. 5. Click-points on a class icon to aid navigation.

Clicking on a *class views* point provides a list of views this class is a member of. Similarly, a *feature views* point provides a list of views a feature occurs in. A view can be selected from a views list dialogue and it will then become the new current view with its window brought to the front. *Class text* points select a default textual view a class occurs in to become the current view. If the class does not yet have any text view, one will be created and will become both the current view and the default text view for the class. Similarly, *feature text* points select or create the default text view for a feature. Clicking on a *class features* point provides a list of all features defined for a class. Clicking on an *all class features* point provides a list of all features for the class, including inherited features, in alphabetical order. Any feature may be selected from these lists and its views or default text view made the current view. Features and other class information can also be expanded into a view using these dialogs (either as part of a class icon or with extra class icons and glue added).

SPE provides menus for textual views that perform similar facilities to click points. Any elements the text of which has been selected with the mouse can provide dialogs with feature lists or be manipulated like graphical icons (e.g. be hidden or their base data removed from the program).

Programmers can construct additional views for the sole purpose of program browsing, based on information selected from other views. This provides a very flexible static program visualisation mechanism with view composition and layout under the complete control of a programmer. Textual views of a class can also be constructed showing a subset of a class’s entire interface.

## 6. View Consistency and Update Records

Object-oriented software development tends to be an evolutionary process [Henderson-Sellers and Edwards 1990; Coad and Yourdon 1991]. Hence program design and implementation may require change for a variety of reasons:

- The analysis of a program is changed affecting design and implementation.
- A design may be incomplete and requires modification impacting on its implementation.
- Errors are discovered on execution, correction of which may result in design changes.

"Changes" may even be transient in that they inform programmers of tasks to perform or errors requiring correction. Many CASE tools and programming environments provide facilities for generating code based on a design [Coad and Yourdon 1991; Wasserman and Pircher 1987] but few provide consistency management when code or design are changed.

### *6.1. Update Records*

A change in SPE may impact on more than one area of a program and may affect more than one view. Thus a mechanism is needed for managing changes, maintaining consistency between views after a change, and recording the fact that change has taken place. SPE uses *update records* for these purposes. When a change takes place (for example, a feature is renamed or its type changed, or a generalisation relationship is added to or removed from a class) this modification is recorded as an update record against the class and possibly some of its sub-components.

drawing_window-Class Definition

```
/*updates_start(94).
update(32). % rename feature figures to gfigures
update(36). %  add client/server design call clicked :
-> figure : pt_in_figure
update(44). %  *** Compilation Error: Duplicate feature
names for clicked
updates_end. */

/*
 * Drawing Window class.
 *
 */

class(drawing_window,
 parents([
  window([rename(clicked,window_clicked)])
 ]),
 features([
  buttons:list(drawing_button),
  current_button:drawing_button,
  figures:list(figure),
  clicked,
  shift_clicked,
  add_figure,
  remove_figure,
  duplicate,
  find_rectangles,
  create_figure,
  figure_rectangle,
  figure_oval,
  oval_to_rectangle,
  rectangle_to_oval,
  change_figure,
  clicked
 ])).
```

drawing_window-Class Definition

```
/*updates_start(94).
update(36). %  add client/server design call clicked :
-> figure : pt_in_figure
update(44). %  *** Compilation Error: Duplicate feature
names for clicked
updates_end. */

/*
 * Drawing Window class.
 *
 */

class(drawing_window,
 parents([
  window([rename(clicked,window_clicked)])
 ]),
 features([
  buttons:list(drawing_button),
  current_button:drawing_button,
  gfigures:list(figure),
  clicked,
  shift_clicked,
  add_figure,
  remove_figure,
  duplicate,
  find_rectangles,
  create_figure,
  figure_rectangle,
  figure_oval,
  oval_to_rectangle,
  rectangle_to_oval,
  change_figure,
  clicked
 ])).
```

Fig. 6. (a) Updates expanded in a textual view, (b) first update applied.

### *6.2. Propagation of Update Records*

Update records are propagated from the view which is the source of a modification to the base view and, from there, to all other views affected by the modification. SPE, however, does not necessarily modify these other views automatically. For example, when a textual view is selected, any updates on elements in the view are expanded in a human-readable form, giving the programmer an opportunity to review the effects of changes made to other views. The programmer can have SPE apply the update to the view, can manually implement the update, or can reject the update. In the latter case, reparsing the view will effectively undo

the change with the effects being propagated to other affected views. Changes are expanded into any textual view, including documentation views.

Fig. 6 (a) shows a textual view with two update records expanded after corresponding changes in graphical views. The `updates_start` comment associated with all text forms is used as the position to expand the update records. The first update is a change of the name of the *figures* feature to *gfigures*. The second is the addition of a client-supplier relationship indicating that method *clicked* calls method *pt_in_figure* of class *figure*. Fig. 6 (b) shows the result after the programmer requests the first update to be applied to the view; the update record has been removed from the view and the feature's name has been changed appropriately.

Some changes can not be directly applied by SPE to a textual view. SPE can not automatically make a change for adding or removing a client-supplier connection such as the second update record in Fig. 6 (b). This is because such a connection in a textual view is implemented as a feature call whose arguments and position in the Prolog code can not be determined. The update is expanded, but the programmer is expected to make an appropriate change to the Snart code and remove the update record.

Update records are also used for processing of errors. For example, during compilation of class data, any semantic errors are expanded into textual views using update records. Fig. 6 (b) also shows such an example, where two features of the same name have been defined in a class.

For graphical views, updates from textual manipulation and parsing or other graphical views are reflected by making the change directly to the icons in the view. If an aspect of a program has been deleted (for example, a feature moved to a sub-class), any inconsistent feature connection is drawn shaded or coloured to indicate the deletion.

*6.3 Update histories*

Update records provide more than a consistency mechanism. Update records for a program component may be viewed via a menu option, providing a persistent history of program modification. User-defined updates may also be added to document change at a high level of abstraction. Programmers may add extra textual documentation against individual updates to explain why the change was made and possibly who made it and when. Fig. 7. shows the dialogs used to view updates and add user defined updates.

*6.4 Integrated Design Using Update Records*

Modifications to a program can be made at any level (analysis, design or implementation) and to any view. Update records will record the change and propagate it to any other affected views. This produces a very integrated environment with little distinction being made between graphical or textual program manipulation. In fact, little explicit distinction is made between the different phases of software development, unlike other systems with different tools being employed for different phases of development [Wasserman and Pircher 1987]. For example, if the drawing program requirements are extended so that wedge-shaped figures and arbitrary polygon figures are supported, these changes are made incrementally at each stage. Analysis views are extended to incorporate new figure and button classes, and new features are added to classes. Design-level views are extended to support the requirements of each

new type of figure and implementation-level views are added or modified to implement these changes.

Update records and the update history could also form the basis of a very flexible version control system. This would allow arbitrary undo/redo of updates (and groups of updates). Update records could also moderate concurrent access and modification of shared program views being developed by multiple users. We are extending SPE to utilise update records for these and other large-scale software development purposes.

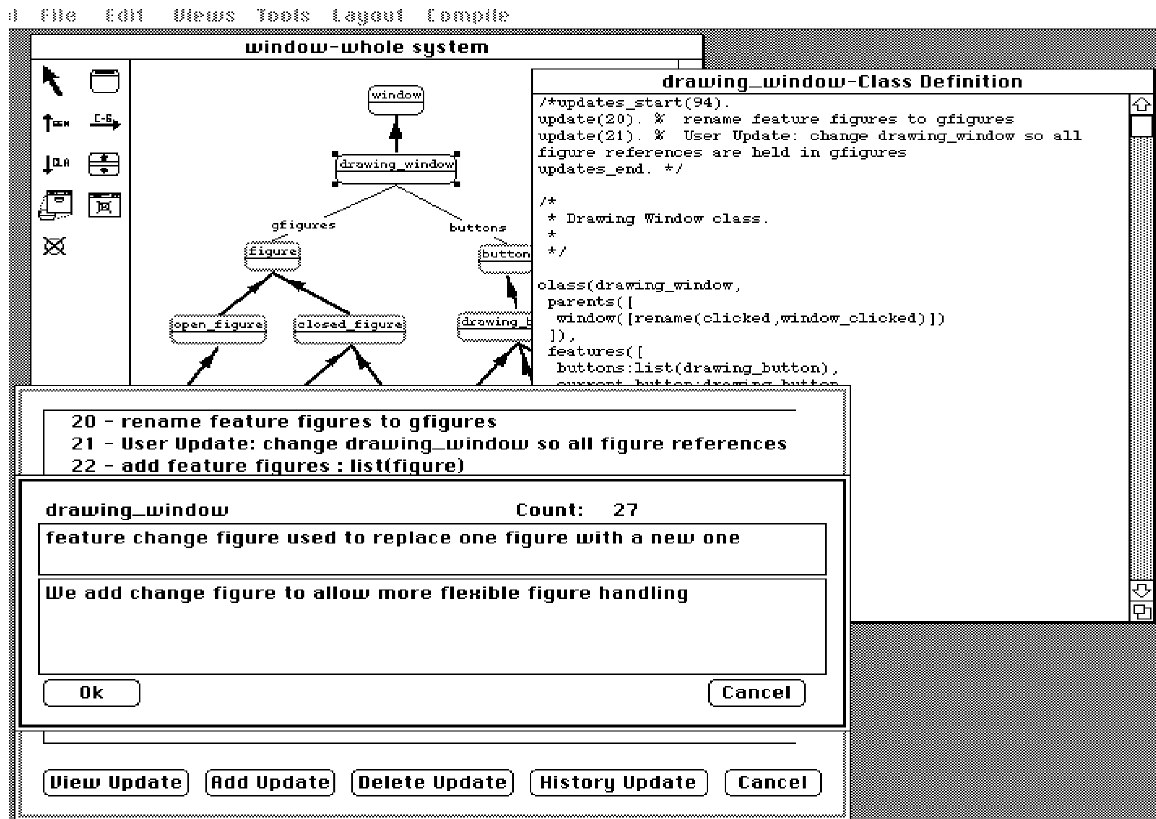


Fig. 7. User defined updates and update viewing.

## 7. Run-time Support

After update, textual views are parsed and the base program information updated to reflect any changes. Semantic values such as the entire interface for a class are recomputed when required. SPE uses the existing compiler and run-time system for a language to generate executable code and run a program. For example, after recompiling a textual view, SPE calls the Snart compiler with the textual view contents to regenerate its method dispatch tables and method code.

SPE provides dynamic visualisation views that display the state of run-time objects. The existing Prolog debugger is used to trace control flow between methods and object views to browse the state of objects. Fig. 8. shows SPE being used to debug the drawing program. We are working on extending SPE's object visualisation capabilities to include graphical object

diagrams [Fenwick and Hosking 1993] and provide data structure and control flow visualisation, similar to [Haarslev and Möller 1990].

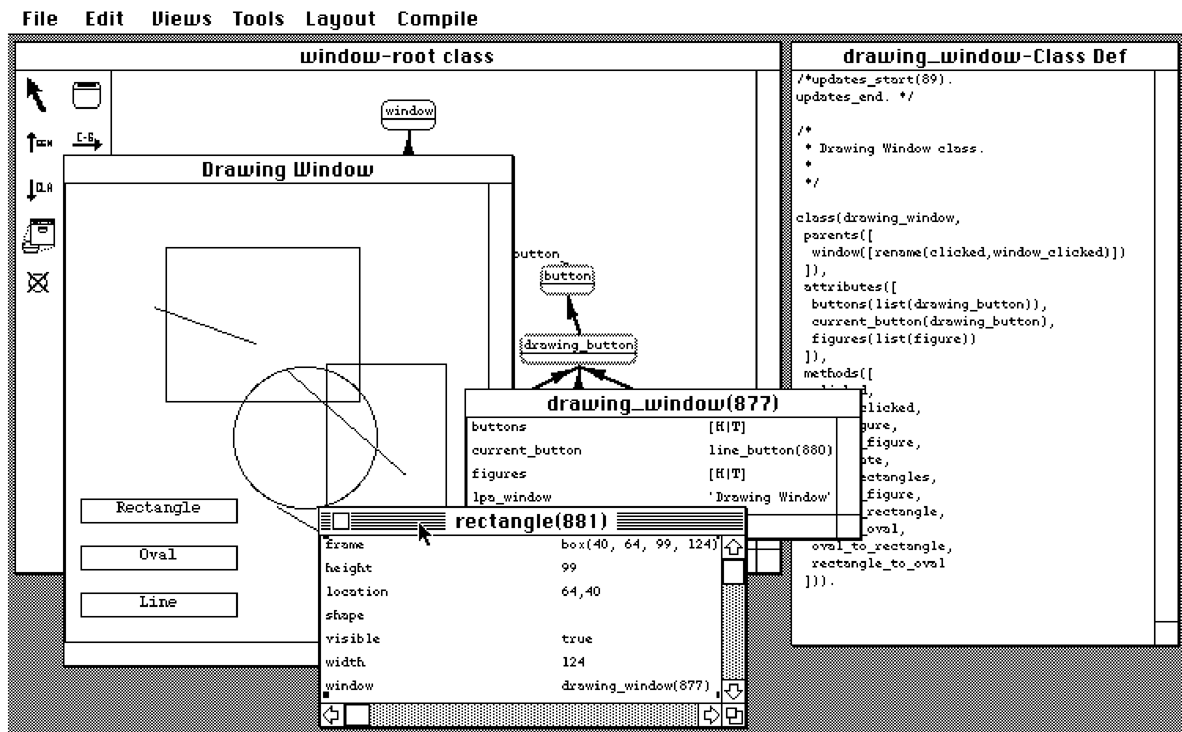


Fig. 8. Debugging the drawing program in SPE.

## 8. The MViews Framework

SPE is implemented as a specialisation of MViews, a generic framework for developing multiple-view based programming environments [Grundy and Hosking 1992 and 1993]. MViews provides base support for: multiple textual and graphical views; consistency management via the update record mechanism; undo and redo facilities using update records; and persistence.

To produce a reusable architecture for MViews, a programming environment (PE) generator with its own specification language could have been constructed, similar to that of the Synthesizer Generator [Reps 1984], or a specialisable framework implemented, as used in Unidraw [Vlissides 1990]. However, many aspects of a good, interactive PE, such as the editor functionality and tool interfacing systems, require specialisation and fine-tuning on a scale difficult to provide with a specialised PE generator. We also wanted to experiment with various aspects of MViews and SPE which is difficult to do with a specialised generator language assuming an unchanging formal basis. For these reasons, the second approach was chosen and MViews was implemented as an extensible object-oriented framework (using Snart as the implementation language). The object-oriented approach taken to the design of MViews simplified considerably the implementation of facilities such as undo-redo, and update record propagation.

Specialisation of MViews to SPE is in two steps. IspelM is a generic specialisation of MViews for object-oriented programming. It provides most of the graphical tool support used

by SPE, together with other language-independent facilities. SPE specialises IspelM further for programming in Snart, through provision of Snart-specific facilities, such as parsers and unparsers. Specialisation of IspelM to support other object-oriented languages is possible, as is the tailoring of SPE or IspelM to support alternative notations for program elements compatible with the programmer's preferred OO Analysis/Design methodology.

## 9. Conclusions and Future Research

We have described SPE, an environment for object-oriented programming. Multiple textual and graphical views combined with a consistency management system based on update records provides integrated support throughout the program development life-cycle. High-level conceptual views of a program can be rapidly constructed using the graphical tools, either as a means of constructing the program (visual programming) or to understand, document, or browse the program. More detailed information, including detailed documentation, can be added using the various types of textual view. Modifications are propagated between views using the record update mechanism, which also acts as a modification history. View support extends to program execution through the object viewer facility and the, as yet rudimentary, program visualisation views.

Current work is aimed at providing multi-user program construction support (including concurrent programming moderated by update records), version control and macro editing operations using update records, and improving execution time support facilities, based on preliminary work by [Fenwick and Hosking 1993].

The MViews framework underlying SPE has application beyond the realm of OO programming environments. Current projects using MViews as a base include a multiple-view entity-relationship diagrammer, dataflow language methods for SPE, and a tool for providing multiple views of a building model for use in computer integrated building construction [Amor and Hosking 1993].

## Acknowledgments

We gratefully acknowledge the helpful comments of our colleagues Rick Mugridge and Robert Amor and the financial support of the University of Auckland Research Committee. John Grundy has been supported by an IBM Postgraduate Scholarship, a William Georgetti Scholarship and a New Zealand Universities Postgraduate Scholarship while pursuing this research.

## References

Amor, R.A., Hosking, J.G., 1993: Multi-disciplinary views for integrated and concurrent design, submitted to *Management of Information Technology for Construction*, First International Conference, Singapore, August 1993.

Arefi F., Hughes C.E., and Workman D.A. 1990: Automatically Generating Visual Syntax-Directed Editors, In *CACM* 33 (3), 349-360.

Booch, G. 1991: *Object-Oriented Design with Applications*, Menlo Park, CA, Benjamin/Cummings.

Coad, P., Yourdon, E., 1991: *Object-Oriented Analysis*, Second Edition, Yourdon Press.

Cox, P.T., Giles, F.R., Pietrzykowski, T. 1990: Prograph: a step towards liberating programming from textual conditioning, *1990 IEEE Workshop on Visual Languages*, IEEE, 150-156.

Fenwick, S.P., Hosking, J.G. 1993: *Visual Debugging of Object-Oriented Systems*, Departmental Report No. 65, Computer Science Department, University of Auckland.

Fischer, G. 1987: Cognitive View of Reuse and Redesign. *IEEE Software*, July 1987, 60-72.

Grundy, J.C., Hosking, J.G., and Hamer, J. 1991: A Visual Programming Environment for Object-Oriented Languages, *Proc TOOLS US '91*, Prentice-Hall, 129-138.

Grundy, J.C., Hosking, J.G. 1992: MViews: A Framework for Developing Visual Programming Environments, *Proc TOOLS Pacific '92*, Prentice-Hall.

Grundy, J.C., Hosking, J.G. 1993: The MViews Framework for Constructing Multi-view Editing Environments, to appear in *New Zealand Journal of Computing*.

Haarslev, V., Möller, R. 1990: A Framework for Visualizing Object-Oriented Systems, *Proc OOPSLA '90*, 237-244.

Henderson-Sellers, B. and Edwards, J.M. 1990: The Object-Oriented Systems Life Cycle. *CACM*, 33 (9), 142-159.

Ingalls, D., Wallace, S., Chow, Y.Y., Ludolph, F., Doyle, K. 1988: Fabrik: A Visual Programming Environment, *Proc OOPSLA '88*, 176-189.

Magnusson, B., Bengtsson, M., Dahlin, L., Fries, G., Gustavsson, A., Hedin, G., Minör, S., Oscarsson, D., Taube, M. 1990: An Overview of the Mjølner/ORM Environment: Incremental Language and Software Development, *Proc TOOLS '90*, Prentice-Hall.

Minör, S. 1990: *On Structure-Oriented Editing*, PhD Thesis, Department of Computer Science, Lund University, Sweden.

Ratcliff, M., Wang, C., Gautier, R.J., Whittle B.R. 1992: Dora - a structure oriented environment generator, In *Software Engineering Journal*, 7 (3), 184-190.

Reiss, S.P., 1985: PECAN: Program Development Systems that Support Multiple Views, *IEEE Transactions on Software Engineering*, 11, 3, 276-285.

Reiss, S.P., 1986: GARDEN Tools: Support for Graphical Programming, *Lecture Notes in Computer Science #244*, Springer-Verlag, 59-72.

Reps, T. and Teitelbaum, T., 1984: The Synthesizer Generator. *Proc of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, New York, 42-48.

Vlissides, J.M., 1990: *Generalized Graphical Object Editing*, PhD Thesis, Stanford University, CSL-TR-90-427.

Wasserman, A.I., Pircher, P.A. 1987: A Graphical, Extensible, Integrated Environment for Software Development, *SigPlan Notices*, 22 (1), 131-142.

Wasserman, A.I., Pircher, P.A., Muller, R.J. 1990: The Object-oriented Structured Design Notation for Software Design Representation, *IEEE Computer*, March 1990, 50-63.

Welsh, J., Broom, B., Kiong, D. 1991: A Design Rationale for a Language-based Editor, In *Software - Practice and Experience* 21 (9), 923-948.

Wilson, D.A. 1990: Class Diagrams: A Tool for Design, Documentation and Teaching, *JOOP*, January/February 1990, 38-44.