Revised verion is Chapter 11 in: Visual Object-Oriented Programming, M. Burnett, A. Goldberg, T. Lewis Eds, Manning/Prentice-Hall, January 1995.

Connecting the pieces: integrated development of object-oriented systems using multiple views

John Grundy[*] , John Hosking, Stephen Fenwick[†] , and Warwick Mugridge
Department of Computer Science
University of Auckland, Auckland, New Zealand

ABSTRACT

SPE is an environment for developing object-oriented software. It allows a programmer to construct multiple visual and textual views of a program, each of which can be edited. All views are kept consistent through an underlying change propagation mechanism. Using this mechanism, together with tools which permit simple flow between phases, SPE encourages an evolutionary approach to object-oriented software development. Cerno is a complementary debugger/dynamic program visualisation package, developed using the same underlying architecture as SPE. Acting together, the two systems provide a very flexible and fluid environment spanning all phases of software development.

1. Introduction

The aim of this work is to provide an integrated, yet extensible, environment for constructing object oriented (OO) software. Many useful tools have been developed to support various parts of the OO development cycle. There is, however, a need for a suitable collection of tools that not only provide support for each phase of software development but also communicate with each other in a transparent manner [Meyers 1991]. We believe that such a collection requires an environment supporting multiple visual and textual views, each of which can be used to either visualise or construct a program, and which are kept consistent with one another in as automatic a fashion as possible.

In this paper, we describe SPE (the Snart Programming Environment), developed to meet the above aim. SPE provides an environment for programming in Snart, an object-oriented extension to Prolog [Grundy 1993]. It should be stressed that although SPE is tailored for Snart, the ideas are applicable to OO software development in general. In particular, the notations used are not as important as the general ideas of consistent multiple visual and textual view support.

The paper commences with a discussion of desirable features of an integrated software development environment for object oriented systems, before reviewing existing work in this area. This is followed by a description of SPE, and its approach to fulfilling these features. We then describe Cerno, a counterpart to SPE that provides execution-time support, before discussing current work and conclusions.

2. Desirable features of an integrated environment

[*] Now at Department of Computer Science, University of Waikato, Hamilton, New Zealand.
[†] Now at Department of Computer Science, Australian National University, Canberra, ACT, Australia.

For an individual programmer, a useful environment capable of supporting integrated OO software development should include at least the following features:

- *Support for visual representations:* It is hard to imagine constructing or maintaining an OO program without diagrammatic aids, even if these amount to simple inheritance graphs. The various competing OO analysis and design methodologies, e.g. [Booch 1991], [Coad and Yourdon 1991], [Henderson-Sellers and Edwards 1990], or [Wasserman et al 1990], provide a wealth of diagrammatic techniques and notations to support OO development. These support, amongst other things, depictions of structural relationships, such as inheritance and aggregation graphs, through to depictions of more dynamic relationships, such as call graphs, dataflow graphs, scenario diagrams, and "timing" diagrams. Variations of these are also available to support analysis or overview documentation, through to detailed low level design, implementation, and debugging.

- *Support for textual representation:* While visual representations are good for expressing high level relationships, it is the authors' opinion that text is better for representing program detail compactly yet understandably. For example, the authors prefer a textual style for representing expressions over visual approaches such as Prograph's dataflow graphs [Cox et al 1989] or forms-based approaches such as that used in Forms3 [Burnett and Ambler 1994]. Documentation too, requires a mixture of textual and diagrammatic forms. In this regard, we are not "radically visual", to use the terminology of [Furnas 1993], but acknowledge that different programmers have different "visual literacy". Hence environments should support programmers in using whatever representations they feel comfortable with, and in whatever way.

- *Support for both visualisation and construction:* Different visual and textual representations should be available to both visualise OO programs and to visually or textually construct them. For example, drawing an inheritance arc in an inheritance diagram should create that inheritance relationship in the program. Support for construction at a variety of levels is needed: analysis and design tools help "construct" abstract program entities, to be refined into more concrete entities using design and implementation tools.

- *Views must be consistent:* Each *view* of a program, i.e. a diagrammatic or textual representation of part of the program, must be consistent with all other views [Meyers 1991]. A corollary of the discussion about "visual literacy" above is that it is undesirable to have any one view type as the "master view" with other views being for visualisation only. Thus all types of program view should be editable in ways sensible to their semantics, with any changes propagated to other affected views. However, there may be only partial mappings between views. This is particularly likely in the earlier phases of program construction, and when transferring between phases, where more detail is needed to suitably implement an analysis or design decision. This partial consistency problem is an important one that must be addressed by any integrated environment.

- *Many views of software systems are needed:* An environment must support many views of the software under development. This means supporting not only multiple view types, such as inheritance graphs and call graphs, but multiple views of each type. This allows views to be created which focus on limited aspects of the software's structure or functionality, for example just the inheritance relationships between the figure classes in a geometric application. Allowing multiple views can avoid the difficulty of large, unwieldy visual views.

- *View contents and layout must be user controlled:* Users must be able to specify what they want seen in each view. This implies the need for simple ways of selecting (for visualisation) or constructing information to be displayed. Users should also be able to lay out views in the way they want so that, for example, locality in diagrams can be exploited. In addition, users should be able to mix representations, possibly in controlled ways, within each view. Thus, for example, to explain the operation of the rendering services in a drawing application, it may be useful to include elements of call graphs, inheritance graphs, and whole part graphs, together with textual annotations, all on the one diagram. In this latter respect we have similar aims to those of the literate programming community [Knuth 1992], but without the dual straight jackets of a fixed linearisation of the program, and limitation to only textual representation.

- *Phase boundaries should be fuzzy:* Easy flow is needed between phases, given that phases in OO programming are far less distinct than in conventional system development [Coad and Yourdon 1991]. The development process tends to be more of a continuum, with additional notation/syntax available in later phases. The development process also tends to be more evolutionary [Henderson-Sellers and Edwards 1990] [Booch 1991], with many small cycles of analysis-design-implementation, rather than one large one. Environments that impose rigid phase boundaries work against this flexible approach to programming.

- *View navigation must be simple:* Supporting many small views of a program implies a local reduction of complexity, as each view focuses on a small part of the system; but global complexity then becomes a problem. Thus simple and intuitive ways of navigating to any view of relevance plus the ability to have several views displayed at once are needed. A program could be thought of as a hyper-document, with navigation via (automatically constructed) hyperlinks connecting relevant views together. Navigation facilities should also extend to program execution views as well to support rapid debugging and maintenance.

- *The effects of design decisions must be traceable:* It should be possible to trace how requirements are fulfilled by design and implementation decisions and vice versa [Boehm 1984]. The lack of support for traceability is one of the most critical failings of existing CASE tools. Traceability is closely related to the issue of partial consistency noted above: solving the partial consistency problem should provide leverage in traceability support. Traceability, in turn, provides support for impact analysis [Pfleeger 1991]

In addition to the requirements of the individual programmer, tool support is needed for team development of large systems. This includes version control, libraries, code "ownership", security, transaction control and locking, project management, testing, metrics and other collaborative software development tools. In this paper we will focus mainly on the single programmer issues, but will briefly discuss issues of multi-user support and versioning in the section on current work.

3. Existing work

There have been many approaches to the development of integrated development environments for textual programming. These include: the Gandalf project [Habermann and Notkin 1982]; Centaur [Borras et al 1988]; MELD [Kaiser and Garlan 1987]; and Mjølner/ORM environments [Magnusson et al 1990]. While these systems support many of the requirements of the previous section they lack integrated support for visual programming.

CASE tools provide graphical editors supporting the construction of analysis and design diagrams [Chikofsky and Rubenstein 1988]. They usually provide consistency management between different views to ensure a software developer has a consistent view of the software system under construction. Software thru Pictures [Wasserman and Pircher 1987] provides various views which support dataflow analysis, structured analysis and detailed object-oriented design. The OOATool [Coad and Yourdon 1991] supports Coad and

Yourdon Object-Oriented Analysis. TurboCASE [StructSoft 1992] supports entity-relationship modelling, structured analysis and design methodologies, and object-oriented analysis and design methodologies.

Most CASE tools do not support complete program implementation. A common approach is to generate program fragments from a design and allow programmers to incorporate these into their own programs. A major drawback of this approach is the problem of “CASE-gap”, where modifications to the design and/or implementation lead to inconsistencies between them.

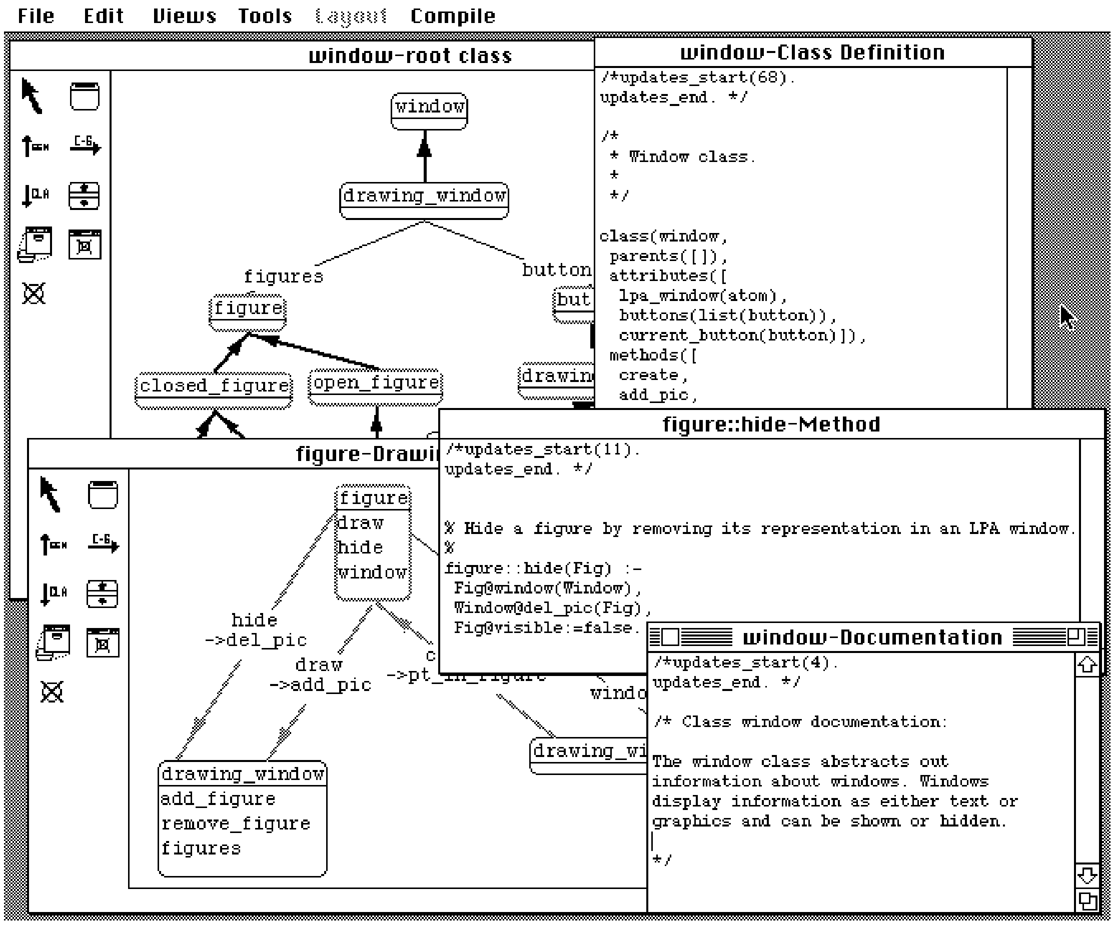
CASE tools are often incorporated into software development environments to provide analysis and design capabilities [Reiss 1990a] [Reiss 1990b]. One problem is that their data storage and user interfaces are difficult to integrate, especially when CASE tools are developed separately from the environment into which they are integrated.

A variety of OO Visual Programming languages exist, which, unlike CASE tools, take the visual paradigm through to program implementation and, in some cases, execution. For example, Prograph [Cox et al 1989] uses dataflow diagrams to specify programs and provides an object-oriented structure in which methods are implemented as dataflow diagrams. An interface builder allows programmers to specify user interface components diagrammatically and specify user interface semantics using dataflow diagrams. Prograph's dataflow diagrams are "reused" during execution to visualise program behaviour. As another example, the Forms languages of [Burnett and Ambler 1994] use a form-based paradigm for visually programming abstract data types. While these languages are important and useful in their own right, and provide very good integration between implementation and execution, none yet provides an "all-phases" solution to software development. In addition, these languages tend to adopt a completely visual approach, at odds with the arguments in the previous section for textual representation support.

FIELD environments [Reiss 1990a] [Reiss 1990b] provide the appearance of an integrated programming environment built on top of distinct Unix tools. Program representation is usually as text files with each tool supporting its own semantics (with conventional compilers and debuggers). Views are not directly supported but tool communication via selective broadcasting [Reiss 1990a] allows changes in one tool “view” (for example, an editor) to be sent to another tool “view” (for example, the debugger or compiler). Free-edited textual program views are supported, but these text views cannot contain over-lapping information. Graphical representations are generated from cross-reference information, but a lack of user-defined layout and view composition for these graphical views is a problem [Reiss 1990b]. Data storage is via Unix text files and a simple relational database (for cross-reference information). Data integration is thus not directly supported but tools can implement translators, driven by selective broadcasting, to allow data from one tool to be used by another.

Dora [Ratcliffe et al 1992] environments support multiple textual and graphical views of software development; all editing is structure-oriented. Dora uses the Portable Common Tool Environment (PCTE) [Wang et al 1992] to store program data and uses PCTE view schemas to provide selective tool interfaces to these programs. Dora supports the construction of analysis and design views, as well as implementation code views, but assumes these views are updated by structure-oriented editing of base program data. It is not clear what the effect is on an abstract, design-level structure when a corresponding code-level structure is updated.

While each of the above systems implements portions of the requirements, there is still a need for a more holistic approach to the integration problem. In the following sections we introduce SPE and its approach to satisfying the requirements of Section 2, based on the mechanism of update record propagation to achieve view consistency.



```
/*updates_start(68).
updates_end. */

/*
 * Window class.
 *
 */

class(window,
 parents([]),
 attributes([
  lpa_window(atom),
  buttons(list(button)),
  current_button(button)]),
 methods([
  create,
  add_pic,
```

```
/*updates_start(11).
updates_end. */


% Hide a figure by removing its representation in an LPA window.
%
figure::hide(Fig) :-
 Fig@window(Window),
 Window@del_pic(Fig),
 Fig@visible:=false.
```

```
/*updates_start(4).
updates_end. */

/* Class window documentation:

The window class abstracts out
information about windows. Windows
display information as either text or
graphics and can be shown or hidden.

*/
```

Fig. 1. Example SPE graphical and textual program views.

4. Views and view types in SPE

Fig. 1. is a screen dump of SPE in use showing two visual and three textual views of a program implementing a simple drawing package. This example illustrates some of the types of view available with SPE. The *window-root class* view shows several of the inheritance and abstract aggregation relationships between various classes. The *figure-Drawing* view shows method-calling protocols related to rendering figures in a drawing window. The *window-Class Definition* and *figure::hide-Method* views are textual views showing a class and method implementation respectively, while the *window-Documentation* view is a textual view serving to document the window class. These views together provide analysis, design, implementation, and documentation views of the drawing package, all within the one environment. The addition of Cerno, described in Section 8, extends this to visualisation of program execution as well.

In the visual views, classes are represented as icons, with textual annotations for the class name, and (a selection of) its features. Various types of, possibly annotated, connectors are used to depict inter-class relationships. For example, bold arrows represent inheritance (eg closed_figure inherits from figure), thin annotated lines represent aggregation relationships (drawing_window has a feature figures which is a collection of figures) and annotated grey arrows represent method calls (eg the hide method of figure calls the del_pic method of drawing_window). The amount of annotation varies depending on the "abstractness" of the relationship: at the analysis level, classes are simply related together in an abstract way; at the design and implementation level, annotations detail how the relationship is implemented. As mentioned in the introduction, the particular notations used are unimportant for the purposes of this paper. SPE could readily be adapted to suit the reader's favourite diagramming notations; tailoring SPE to support other notations or languages is not demanding[1].

Information in a view may overlap with information in other views. For example, the figure and drawing window classes appear in both graphical views, and the drawing window class appears twice in the *figure-Drawing* view. 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, as described in Section 7.

5. View construction and view navigation

The user of SPE may construct any number of views of a program. Fig. 2, for example, shows several additional design level views used in constructing the drawing program example of Fig. 1. Each view may be hidden or displayed using the navigation tools described below. Elements may be added to a view using a variety of tools, with an arbitrary depth undo-redo facility. We distinguish between 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 static 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 element addition. Tools are provided for addition of classes, class features (added to the class icon), and generalisation and client-supplier relationships. For the latter, additional information about the relationship (name, arity, whether it is inherited, and type - call, aggregate, abstract) is specified by dialog box. Class icons can be selected and dragged (using a selection tool) in a similar manner to figures in a drawing package. Tools are also provided for creating new views and removing elements from a view or completely from the program.

Mouse operations on class icons allow views to be extended to include feature names, generalisations, specialisations, and the various forms of client supplier relationships defined in other views. This allows specialised views of an existing or partially developed program to be rapidly constructed, focusing on one or more aspects of that program. The view may then be used to modify the program using the visual programming tools. The *figure-Drawing* view in Fig. 1, for example, has been created to show 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. SPE automatically lays out view extension information, but programmers are able to move these extra objects to suit.

[1]For example, a specialisation of SPE has been developed to provide an environment for constructing models in the object-oriented specification language EXPRESS and its graphical variant EXPRESS-G [Amor et al 94].

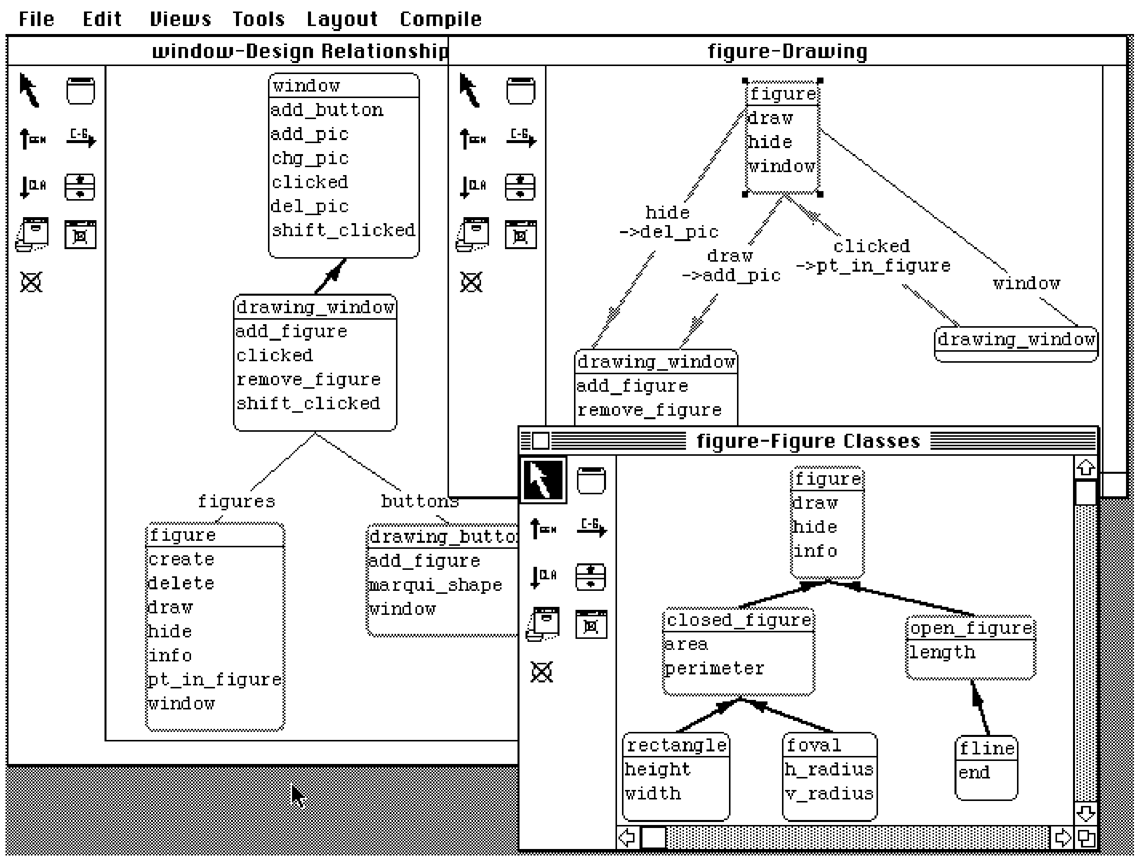


Fig. 2. Examples of design-level diagrams.

Textual views are created in a similar manner to visual views, but consist of one or more *text forms* (class definitions, method implementations, or documentation), rather than icons and connectors. 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[1], and then parsed to update base program information. This contrasts to most other environments, such as Dora [Ratcliff et al 1992], Mjølner [Magnusson et al 1990], and PECAN [Reiss 1985] which use structure-editors for both graphics and text. Structure-edited approaches for text tend to be quite restrictive on the user [Arefi et al 1990] [Welsh et al 1991]. Methods 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, more typically, be generated by extension from other views and then modified in a free-edit fashion. In addition, existing textual programs may be imported into SPE. Textual views for imported programs are automatically constructed, and corresponding visual views can be simply constructed by extension from the textually derived information.

As described in Section 2, programmers need to locate information easily from a large number of views and be able to gain a high-level over-view of different program aspects. SPE's approach is to use the program views themselves in a hypertext-like fashion as the basis for browsing. Class icons in graphical views have a number of active regions, or "click-points", which cause pre-determined actions to be carried out when clicked on (similar to Prograph's dataflow entities [Cox et al 1989]). Click points allow rapid navigation to any other views (textual or graphical) containing the class or individual features of the class. Menus in textual views provide similar functionality to click points, but based on the currently selected piece of text.

[1]Text forms can also be edited using an alternative menu-driven, structure-oriented style of editing.

Programmers can construct additional views, textual or visual, primarily for program browsing, based on information selected from other views.

6. Phases and modes of construction

SPE makes little distinction between the different phases of the design cycle, in keeping with the philosophy of Section 2. There are no special view types for each of the phases. The palette of tools in the visual views can be used to define analysis, design, and implementation level view elements. The one client supplier tool, for example, can be used to specify abstract class relationships, abstract or concrete aggregate relationships, and abstract or concrete method calling protocols by a combination of click and drag mouse operations and auxiliary dialog box. It is left to the user to decide what information is to be placed in each view. Thus, the user may choose to have separate analysis, design, and implementation views, or may mix several phases in the one view. Relationships initially specified as abstract may be refined to more concrete implementations either within the current view, or by creating a new view and specialising the relationships there.

The lack of distinction between tools for use in various phases is a deliberate one. It would be fairly straightforward to define specialised SPE view types for the different design phases, such as in [Wasserman and Pircher 1987], providing limits on what can be specified in each phase. This would, however, run counter to the rather fluid, evolutionary way in which OO systems tend to be developed.

Users are not constrained to program only in a single "mode" (textual or visual). More "visually literate" programmers can construct their programs mostly via visual views. Most of the important inter- and intra-class relationships (eg inheritance, method and attribute declaration, and method call protocols) can be defined visually. Currently, textual views are required only for detailed method implementation and textual documentation. More "textually literate" programmers can develop their programs using textual views, creating visual views by extension for purposes of documentation, browsing, and static program visualisation, and for communication with their more "visually literate" colleagues.

There is, however, some distinction between the different view types. Detailed method implementation may only be done within textual views, as there is no corresponding visual representation[1]. Some of the high level relationships expressible in visual views have no corresponding representation in textual views, other than as comments. Two specialised textual view types are provided in SPE: canonical textual views for class definition and for method implementation. These views provide a textual rendering of the underlying base view of the program. They can be edited and manipulated just like any other textual view but also serve to specify what the language compiler will "see" of the program. They also have a special significance with respect to view consistency, as described in the following section.

7. View consistency and traceability using update records

The requirements, analysis, design and implementation of a program may change many times during its development. Thus a mechanism is needed for: managing changes, maintaining consistency between views after a change, and recording the fact that change has taken place. Many CASE tools and programming environments provide facilities for generating code based on a design [Coad and Yourdon 1991] [Wasserman and Pircher 1987], and some support reverse engineering, such as the C Development Environment [IDE 92], updating design from code. A few provide complete consistency management when code or design are

[1] We are currently implementing a dataflow-style visual tool, similar to Prograph [Cox et al 1989], to also permit the complete visual construction of systems.

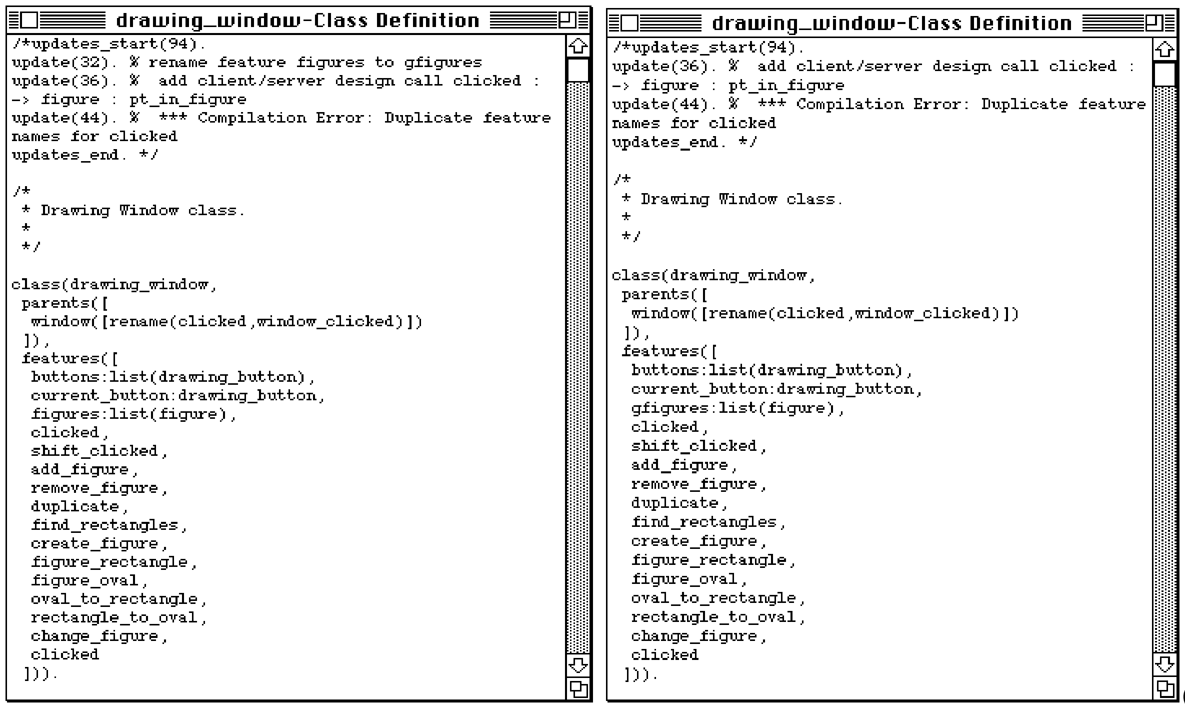
changed, such as Dora [Ratcliffe et al 92], but they can not propagate “fuzzy” updates between design and implementation views (i.e. updates which don’t have a one-to-one correspondence from one level to the next, such as client-supplier relationships).

In addition, the ability to trace requirements and design decisions is important [Boehm 1984] [Pfleeger 1991]. Modifying a requirement implies the need to trace through the design and implementation decisions made to support that requirement and update them appropriately. Similarly, if code implementing a requirement is modified, it is important to be able to trace back to the requirement to check if it is still met.

SPE uses *update records* for consistency management. When a change takes place (for example, a feature is renamed or a generalisation relationship is removed from a class) the modification is recorded as an update record against the class (and possibly some of its sub-components). Update records contain a complete specification of the change made, and form the basis of the undo/redo facility.

Update records are propagated from the view that is the source of a modification to all other views affected by the modification. The views affected by a change are automatically determined from the semantics of the element modified. Thus deletion of a method of a class will cause update records to be propagated to any view containing either the method or the class (and possibly subclasses).

What is done with a propagated update record depends on the view the record is sent to. Consider the case of a method deletion update record. It is ignored if it does not affect the view, for example if the deleted method is not shown in the class' icon. Alternatively, there may be sufficient information for the view to be modified immediately, for example the method name can be removed from a class icon without additional information being required. In this case, however, it is often useful to have a visual indication that a change has occurred as a result of work done in another view. This is done in a number of ways, depending on whether the view is graphical or textual. In SPE visual views, elements deleted by another view are shown "greyed out", to provide the appropriate visual clue. If the user, on reviewing the effect of the change, is unhappy with the change it can be simply reversed by manipulating the greyed out element which acts as an interface to the undo/redo facility. In textual views, the update record is "unparsed" into a human readable form and included as a pseudo-comment in the view. Fig. 3(a), for example, shows a textual class definition view with three update records unparsed at the beginning of the text.



```
/*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
 ])).
```

```
/*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
 ])).
```

(a)

(b)

Fig. 3. (a) Updates expanded in a textual view, (b) first update applied.

As a special case, canonical textual views display every update record prior to its "implementation", so that the user has a final chance to review the changes before "committal". Thus, the first update record in Fig 3(a) indicates that the *figures* feature of the *drawing_window* class has had its name changed (in another view) to *gfigures*. The user may "commit" the changes by selecting the update record(s) and requesting the changes to be effected. Fig 3(b) shows the same view after automatically effecting the first update record.

In many cases, however, there is insufficient information for the view to be updated adequately as there is only a partial mapping between representations. For example, the second update record in Fig 3(a) indicates that there has been an addition of a method call client-supplier relationship to a visual view. This addition contains insufficient information to directly modify the implementation. Missing are parameter information, and the exact location within the calling method where the call is to be made. Similarly, a compilation error record (as shown in the third update record of Fig. 3(a)) does not contain sufficient information to automatically infer the appropriate correction. In these cases, the update records serve to show that the view/program is (potentially) inconsistent or erroneous, and that the programmer may need to make an appropriate change. These "fuzzy" update records are an important and novel contribution of the work presented here. Recognising that automatic consistency is impossible, these update records provide a means for the environment to interact with the creative element (the programmer) responsible for achieving complete consistency.

A special case of "fuzzy updates" occurs in textual documentation views. These receive and display every update record relating to the program entity (class or method) being documented, providing a partially automatic documentation revision facility. The textual contents of the update records can be cut and pasted by the programmer into the body of the documentation in an appropriate way to complete the documentation revision.

SPE's view consistency mechanism provides support for the evolutionary, iterative refinement approach to software development typical of OO programming. 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. At each stage update records are used to notify views of changes that have been, or need to be made, to keep the entire design consistent. Similarly, if changes are made to an implementation or design view, update records are propagated to relevant design and analysis views, thus supporting bidirectional consistency between programming phases.

Update records provide more than a consistency mechanism. Update records are persistent, providing a permanent history of program modification, and may be viewed, for any program component, via a menu option. User-defined updates may be added to document change at a high level of abstraction. Users may also textually annotate an individual update, such as to explain why the change was made, by whom, and when. This mechanism provides documentation support additional to the automatic inclusion of "fuzzy updates" in documentation views.

The update record mechanism currently provides only partial support for tracing requirements through design and implementation (and vice versa). Yet to be implemented is a mechanism whereby changes manually made as a result of an update record can be associated with that update record. This is conceptually fairly straightforward, but requires careful implementation, as such changes may be spread across a number of views and places within a view. Such an extension, together with the ability to navigate via the persistent update records would give full support for traceability.

8. Bridging the gap to execution - Cerno

Multiple program views, at different levels of abstraction, are also beneficial at execution time. Cerno is a visual debugging system for the Snart language which provides multiple views of an executing Snart program [Fenwick et al, 1994]. Each view is kept consistent with the underlying program execution state. A Cerno view is composed of a number of "displays" for objects, together with connectors representing relationships between the objects, such as object references. A display is an iconic depiction of the state of an object or a higher level abstraction of a collection of objects. The contents of a view and the way in which the view is laid out are under the user's control.

Fig. 4 shows Cerno in use together with SPE debugging/visualising the drawing program. In the background is an SPE view. In the foreground is a drawing window, generated by the application, and two Cerno views, one focussing on the button objects and one on the figure objects associated with the drawing window. As with SPE, a Cerno view includes a tool palette (the array of iconic buttons to the left of the Cerno views). Tools include an icon layout editor, an object value editor to modify the underlying program execution state, and a view creation tool. Additional functionality is provided via menus.

The form and contents of a display can be quite flexibly defined and layed out. Filters select specific attributes to display or those that satisfy specified criteria. Layout of information can be done using predefined object view types (such as a vertical list of features plus values, or a bar graph for collections of numbers). New filters and view types can be defined using a simple, but powerful and extensible, icon layout language.

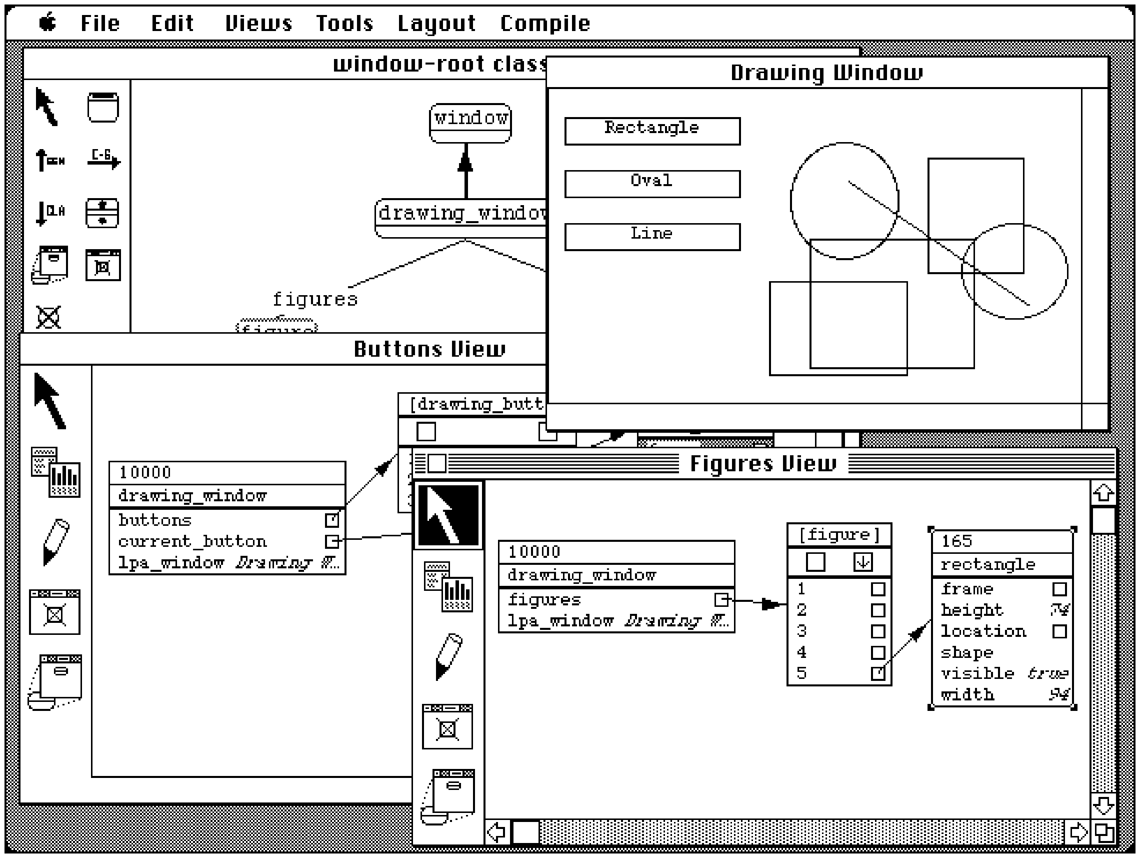


Figure 4: Cerno in use with SPE debugging the drawing program

Rather than accessing and manipulating the underlying object state directly, Cerno's display level interacts with a layer of "abstractors", which, in turn, interact with low level object tracing facilities[1]. Abstractors present a uniform interface to the display level. An abstractor provides a display with a list of (attribute, type, value) triples and informs the display whenever any values in that list change (making use of update records). The display uses the list to construct the iconic representation.

Simple abstractors perform a direct translation from the actual object state to the list of triples. More complex abstractors may be constructed to aggregate information from a number of simple abstractors in a similar fashion to the encapsulators of [Noble and Groves 1992]. For example, abstractors for viewing collections abstractly in a variety of forms, eg linked list of nodes, abstracted list of values, etc are provided. Similarly abstractors may be used to disaggregate information from lower level abstractors; for example, Cerno allows records embedded in objects to be viewed as pseudo objects through the use of appropriate abstractors.

Figure 5 illustrates the use of complex abstractors in Cerno. At the top of the view, a box and connector view of a linked list containing integer values is shown. This uses simple object abstractors, one for each list element. The central display is a high level view of the same list showing the list as a sequence of element number, value pairs, abstracting away the list links. The list abstractor gathers information from the individual object abstractors, and presents this to the display level as a single list of (element number, type,

[1] The Snart language provides object spying facilities which allow any object's execution state to be monitored transparently without program code modification. These facilities provide standard breakpoint insertion, spypoints, and single step execution capabilities.

value) triples, which are displayed as shown. The bottom display uses the same high level list abstractor, but the icon display language has been used to construct a display showing two subviews of the list: as a vertical list of element numbers, types and values, and as a vertical bar graph of values. This complex list view is specified with a line of icon layout language code: (*horizontal([features(long,all),vline(1),features(vertical_graph(-10,10,100),all)])*).

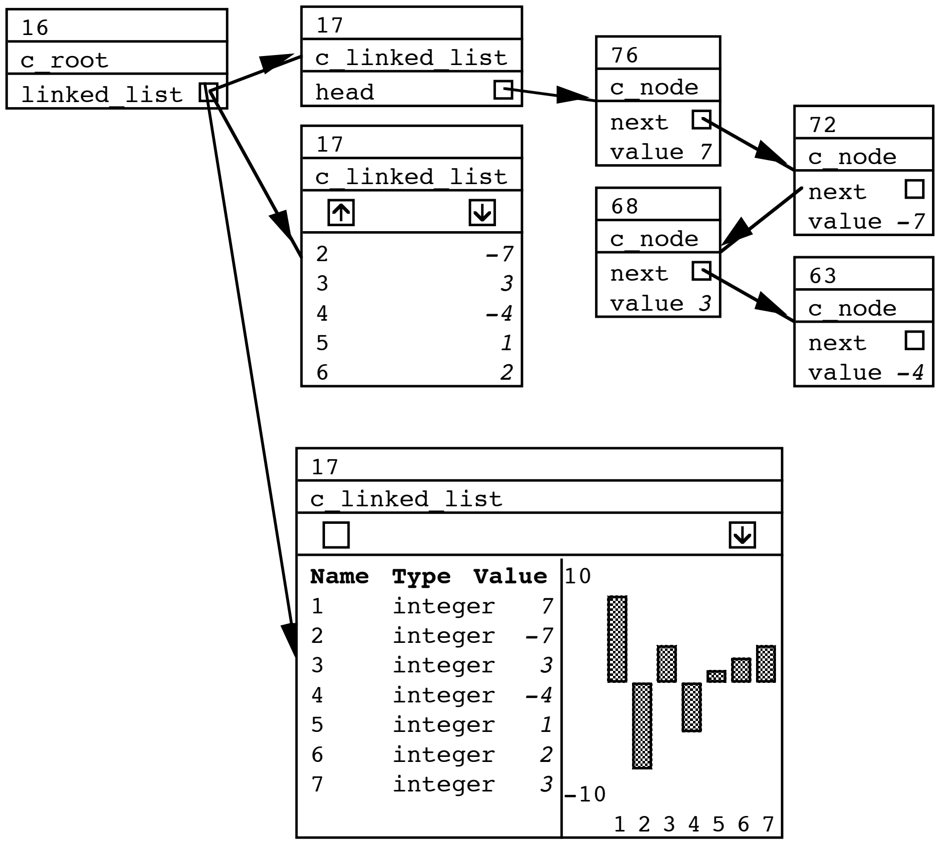


Figure 5: Three Cerno views of a linked list

Abstractors can also be used to provide views of the dynamic calling structure of an executing program. Fig. 6, for example, shows object displays which include both the data relationships between objects and the current method stack. Active methods listed at the bottom of each object display (with vertical bars at the left to distinguish them from attributes) while method calls are shown as arrows connecting the methods. Fig. 7 shows a "timing diagram" (similar to those of [Booch 1991]), visually illustrating the history of method calls during execution of a program. This display makes use of a "history" abstractor, which gathers information from each method call, and provides it to a display view to depict the "time and duration"[1] of each method call with bar graphs.

[1] In terms of the number of other method calls made between the call and return of a method.

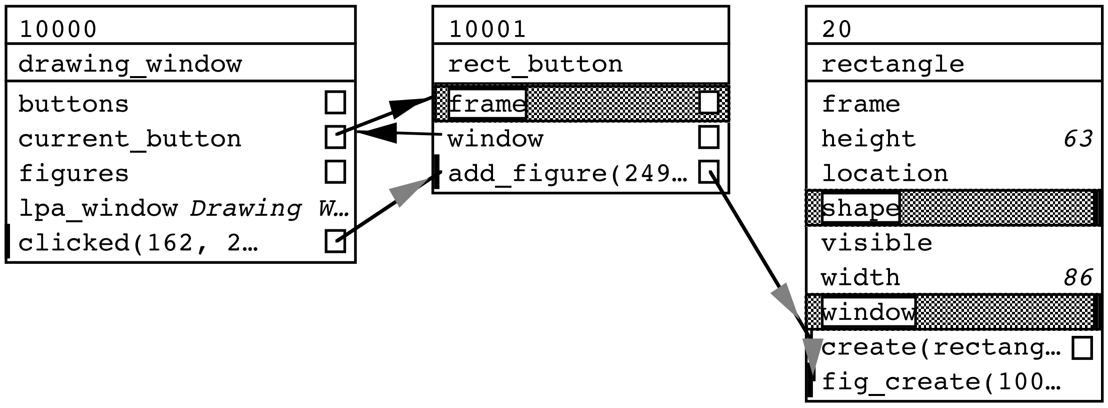


Figure 6: Cerno View showing object references (dark arrows) and active method calls (grey arrows). The hashed background on some attributes indicate their value is incompatible with their declared type.

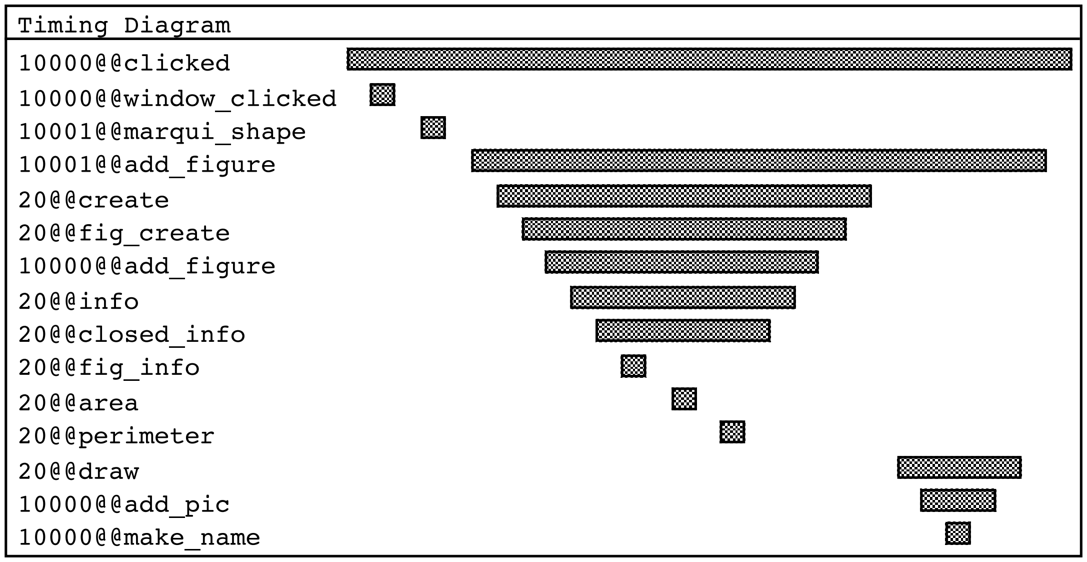


Figure 7: Cerno "timing" diagram, showing "time" and "duration" of method calls (listed as *objectid@@methodname* labels to the left). The "time" is in terms of numbers of method calls, and increases to the right.

Interesting program views can be saved as “templates”, to be reused during later executions of the same (or a similar) program. When applied to an object, a template attempts to reconstruct as similar a view as possible to the view used to construct the template. The views will differ, however, owing to the different objects, object references, and attribute values involved. In addition to their (re)use in fault finding, templates allow execution time views to be reused for dynamic documentation; a program, template and suitable test data can be used to illustrate important aspects of the program's behaviour.

Currently, SPE and Cerno interact in a simple manner: it is possible to navigate from Cerno views to SPE views and vice versa. This allows the definitions of the classes of an object (and related class diagrams) or method implementations to be accessed directly from the object, and for execution views and templates related to a class to be accessed from a class view. Both of these are done in a similar hypertext style manner to the SPE inter-view navigation capabilities. Current work aimed at improving the integration is described in Section 11.

SPE already provides a fairly fluid program development environment, permitting boundaries between different phases of software development to be easily bridged. The integration of Cerno and SPE extends this to include debugging and dynamic program visualisation, providing "cradle to the grave" support for software development.

9. Implementation - MViews

Both SPE and Cerno are implemented as specialisations of MViews, a generic OO framework for developing multiple-view based programming environments described in [Grundy and Hosking 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 program and view persistence. MViews is itself implemented in Snart, the object-oriented Prolog that SPE provides a programming environment for. Snart's combination of object-orientation and logic programming provides a powerful prototyping language, incorporating support for object persistence, object version spaces, declarative and procedural programming styles, and ready access to the powerful graphics capabilities of LPA Prolog [LPA 1992]. Construction of SPE proceeded much faster with Snart than with earlier prototypes of SPE that used Eiffel [Meyer 1992] and C++ [Stroustrup 1984].

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, unparsers, compiler integration, and Snart object management.

The approach of using multiple layers of frameworks added extra complexity to the initial construction of SPE. Some careful design, was required to cope with the multi-step specialisations. Building in useful reusability almost always results in additional costs in terms of the time taken to sort out suitable general purpose abstractions. However, we believe the additional costs are worthwhile as they have led to a much more flexible product.
As an indication of the reusability of the base MViews framework, a system for multiple-view entity-relationship diagramming plus relational schema editing required less than 1 person week to design and implement. Extension of IspelM to support other diagramming notations and OO languages is a straightforward task of appropriately specialising the generic framework. Modifying an icon shape to support an alternative notation, compatible with the programmer's preferred OO Analysis/Design methodology for example, amounts to a few lines of code. Adding an additional graphical tool, involves specialisation of the appropriate window class, the addition of a method for tool invocation and possibly redefinition of generic routines for mouse handling, line drawing, etc.

SPE makes use of a pre-existing language compiler for Snart[1]. Modification of SPE to support other languages, such as C++ [Stroustrup 1984], mostly impacts on the textual view handling components, as the graphical tools are just as applicable to other (class-based) OOLs. Textual views require parsers and unparsers to match the language syntax, and the availability of an existing C++ compiler to use for semantic processing. The parsers and unparsers can be written directly in Snart (or Prolog), or implemented using Yacc or other parser generators. Additional textual forms may also be required. For example, C++ could usefully make use of ".h" forms, while class contract forms for pre and post conditions and class invariants are desirable for Eiffel [Meyer 1992]. To give an indication of the likely costs involved, the specialisation of

[1] Allowing the environment to make use of an existing compiler has obvious advantages over constructing a purpose-built compiler.

IspelM to support programming in the Express and Express-G textual and graphical object-oriented specification languages [ISO 1992] (together with use of Snart and Cerno to prototype and visualise implementations of the resulting Express schema) took only 2-3 person weeks to result in a usable system [Amor et al 1994].

Cerno too, is fairly generic in its approach, and could be used to visualise other OO execution environments. The principal requirement is access to the underlying object states, with appropriate events indicating change of state. This involves modification of the simple object abstractor classes, but very little in the way of modification is required for the rest of the system (including the higher level abstractors). Modifying Cerno to support new display abstractions involves specialising or creating new abstractor classes (written in Snart), while new display visualisations can be simply developed using the icon layout language.

10. Experience

SPE has, to date, been applied to two substantial projects. In one, two existing object oriented systems were integrated together using high level SPE analysis views to perform the initial schema integration, and design and implementation views to implement the inter-schema mappings [Mugridge and Hosking 1994]. This involved use of SPE's textual import capabilities, to load the environment with the legacy systems, and consequent construction of abstract visual views by extension from the textual information. Use of multiple views to manage the complexity of integrating the two large systems was invaluable in completing this project. In particular, the ability to rapidly create abstract visual views from the original textual program was important for the early stages of schema integration [Mugridge and Hosking 1994].

Both SPE and Cerno have been adapted and used extensively in a project for constructing object oriented models of buildings (SPE), using Snart to represent the building schema, and navigable and editable instances of instances of the models (Cerno) [Amor and Hosking 1993]. Here, the view consistency provided by SPE is critically important, as it allows views of the building schema tailored for different building professionals to be kept consistent with one another.

SPE has been used to develop and visualise both the MViews framework, and SPE itself [Grundy 1993]. Use of SPE has enabled us to better visualise the inter-relationhips of MViews components and hence it has assisted in both maintaining MViews and SPE and in making design improvements for future versions of these systems. The multiple views of these complex object-oriented frameworks, particularly the provision of graphical and textual views with bi-directional consistency management, has made their maintenance easier. We are able to make design changes via visual programming or textual programming, whichever is most appropriate. More formal user testing of SPE, comparing the ease of development of given systems with and without SPE, is planned for this Summer.

SPE can represent large, complex graphs within one view. A zoom facility is supported which shows an overview of the whole graph structure and allows programmers to zoom in on specific parts of the graph as required. However, we have found multiple views to be a more useful approach to representing and manipulating such complex visual program structures. A large graph is better split into multiple views, with each view focusing on specific classes and class relationships. This allows programmers to more easily understand the complex structures, as many relationships and classes can be removed from a diagram and shown in a separate view. This approach allows for either hierarchical views, as found in many dataflow systems, like Prograph [Cox et al 89], or many overlapping views. This means that the view navigation facilities of SPE are essential to managing these multiple aspects of a complex software system.

The MViews framework underlying SPE has application beyond the realm of OO programming environments. MViews has been used to develop a multiple-view entity-relationship diagrammer, a dialog box painter/editor, and is currently being used to implement a visual-Pascal [Lyons et al 1993] and to implement an environment for parallel programming.

11. Future Work

While there is much to be done to fully meet the requirements of Section 2, a substantial start has been made. Current work is aimed at: traceability support; tools for large scale software development including multi-user program construction support (moderated by update records); version control and macro-editing operations using update records; and at improving the crossover between Cerno and SPE. An interesting extension, which we are currently tackling, is to permit translation from Cerno templates into SPE class diagram views and vice versa. Thus, an interesting execution time view of a program could be translated into a class diagram, useful for documentation purposes or static browsing. Likewise, program design and implementation views can be translated into templates that can be used to view actual program state. The MViews update record mechanism could also potentially be used to maintain a "pointing finger", or similar, approach to highlight the source code of the currently executing method in an appropriate SPE textual view. There is also potential in adapting Cerno's icon layout language for use in defining SPE's icons. Other extensions to Cerno planned include the creation of "replayable" program animations, and experimentation with new higher level abstractors. There is also scope for adding more specialised view types to both SPE and Cerno, such as the dynamic views of [De Pauw et al 1993].

12. Summary

We have described SPE, an environment for object-oriented programming, and its execution time companion Cerno. Together these implement most of the requirements specified in Section 2. In this summary we review those requirements and the means by which SPE and Cerno implement them.

Visual representations of abstract inter-class relationships and more concrete instantiations of those relationships are provided in SPE. Textual representation in the form of class interface, method implementation, and documentation views are supported. The textual and visual views can be used to both visualise parts of the program, by extension from information supplied in other views, and to construct, by direct manipulation and free form editing, new parts of the program. Visual and textual representation of the execution state of the program is provided by Cerno. Cerno views are also editable, and hence can be used to modify the execution state of the program.

All SPE and Cerno views are kept consistent with one another using the update record mechanism. Where automatic view consistency is not possible, such as changes propagated between SPE views showing different levels of abstraction or programming phases, views are visually annotated to indicate that programmer modifications are required to maintain the consistency. This "fuzzy update" support is a particularly novel feature of the work presented here. A special, and important, example of the latter is in the partially automatic revision of documentation views in SPE. The viewable, editable, persistent history mechanism is another useful and novel byproduct of the update record mechanism.

There is no limit to the number of views that can be constructed in either SPE or Cerno, and users are free to choose both the contents and layout of views. This permits users to construct views that focus on either small parts of the system, or to abstract away detail to provide system overview views. Simple view navigation, via automatically constructed hypertext links, allows users ready access to desired information. The access links between Cerno and SPE bridge the gap between program specification and program execution/testing.

Boundaries between programming phases in SPE are deliberately kept vague (or more precisely there is no enforcement of which relationships can be specified in which type of view). This supports exploratory programming and an evolutionary approach to software development. The proposed facilities for mapping between SPE views and Cerno templates will also assist in breaking the barrier between phases.

Tracability is not yet well supported. The mechanism of expansion of "fuzzy" update records as readable text provides partial support, but extra facilities are required to associate update records propagated from changes made in early phase views with implementations of those changes made in later phase views. A navigation facility using this tracability information will also enhance software development which uses visual techniques. Work is in progress to provide such facilities.

Acknowledgments

We gratefully acknowledge 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. Stephen Fenwick has been supported by a University of Auckland Graduate Scholarship. John Hosking acknowledges the assistance of the EEE Department of Imperial College of Science Technology and Medicine in the preparation of this manuscript.

References

[Amor and Hosking 1993] R.A. Amor and J.G. Hosking, "Multi-disciplinary views for integrated and concurrent design," in *Selected (refereed) papers from the Proc of the First International Conference on Management of Information Technology for Construction* (K.S. Mathur, M.P. Betts, and K.W. Tham, eds), Singapore, 1993, pp. 255-267.

[Amor et al 1994] R. Amor, G. Augenbroe, J. Hosking, W. Rombouts, J. Grundy, *Directions in Modelling Environments*, Technical report, Technical University of Delft, Department of Civil Engineering, 1994.

[Arefi et al 1990] F. Arefi, C.E. Hughes and D.A. Workman, "Automatically Generating Visual Syntax-Directed Editors," *CACM*, Vol. 33, No. 3, 1990, pp. 349-360.

[Boehm 1984] B.W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE Software*, Vol. 2, 1984, pp. 75-88.

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

[Borras 1988] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, "CENTAUR: the system," *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*.

[Burnett and Ambler 1994] M. Burnett and A. Ambler, "Interactive Visual Data Abstraction in a Declarative Visual Programming Language", *JVLC*, March 1994.

[Chikofsky and Rubenstein 1988] E.T. Chikofsky and B.L. Rubenstein, "CASE: Reliability Engineering for Information Systems," *IEEE Software*, Vol. 5, 1988, pp. 11-16.

[Coad and Yourdon 1991] P. Coad, and E. Yourdon, *Object-Oriented Analysis*, Second Edition, Yourdon Press, 1991.

[Cox et al 1989] P.T. Cox, F.R. Giles, T. Pietrzykowski, "Prograph: a step towards liberating programming from textual conditioning," *Proceedings 1989 IEEE Workshop on Visual Languages*, 1989, pp. 150-156.

[De Pauw et al 1993] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides, "Visualizing the behaviour of object-oriented systems," *Proceedings OOPSLA 93*, Washington DC, September 1993, pp. 326-337.

[Fenwick et al 1994] S. Fenwick, J.G. Hosking, and W.B. Mugridge, "Cerno-II: a program visualisation system", Report No. 87, Department of Computer Science, University of Auckland, NZ.

[Furnas 1993] G. Furnas, "Towards radically visual computation," *Proceedings 1993 IEEE Visual Languages Symposium*, Bergen, Norway, 1993, p. 2.

[Grundy 1993] J.C. Grundy, *Multiple textual and graphical views for interactive software development environments*, PhD thesis, Department of Computer Science, University of Auckland, Auckland, New Zealand, 1993.

[Grundy and Hosking 1993] J.C. Grundy and J.G. Hosking, "Constructing multi-view editing environments using MViews," *Proceedings 1993 IEEE Symposium on Visual Languages*, Bergen, Norway, 1993, pp. 220-224.

[Henderson-Sellers and Edwards 1990] B. Henderson-Sellers and J.M. Edwards, "The object-oriented systems life cycle." *CACM*, Vol. 33, No. 9, 1990, pp. 142-159.

[IDE 1993] Interactive Development Environments, The C Development Environment, 595 Market Street, San Francisco, CA 94105.

[ISO 1992] ISO/TC184 1992 Part 11: The EXPRESS Language Reference Manual in Industrial automation systems and integration - Product data representation and exchange, Draft International Standard, ISO-IEC, Geneva, Switzerland, ISO DIS 10303-11, August.

[Kaiser and Garlan 1987] G.E. Kaiserand D. Garlan, "Melding Software Systems from Reusable Blocks," *IEEE Software*, Vol. 4, No. 4, 1987, pp. 17-24.

[Knuth 1992] D.E. Knuth, *Literate Programming*, University of Chicago, Chicago, 1992.

[LPA 1992] *MacPROLOG Graphics Manual*, Logic Programming Associates, London, 1992.

[Lyons et al 1993] P. Lyons, C. Simmons, M. Apperley, "HyperPascal: Using visual programming to model the idea space," *Proceedings 13th New Zealand Computer Society Conference*, Auckland, New Zealand, August 1993, pp. 492-508.

[Magnusson et al 1990] B. Magnusson, M. Bengtsson, L. Dahlin, G. Fries, A. Gustavsson, G. Hedin, S. Minör, D. Oscarsson, M. Taube "An overview of the Mjølner/ORM environment: incremental language and software development, *Proceedings of TOOLS '90*, Santa Barbara, October 1990, pp. 635-646.

[Meyer 1992] B. Meyer, *Eiffel the language*, Prentice Hall, Herts, United Kingdom, 1992.

[Meyers 1991] S. Meyers "Difficulties in integrating multiple view editing environments," *IEEE Software*, Vol. 8, No. 1, 1991, pp. 49-57.

[Mugridge and Hosking 1994] W.B. Mugridge and J.G. Hosking, "Towards a lazy, evolutionary common building model," accepted for publication in *Building and Environment*., 1994.

[Noble and Groves 1992] K.J. Noble and L.J. Groves "Tarraingim - a program animation environment," *New Zealand Journal of Computing*, Vol. 4, No. 1, 1992, pp. 29-40.

[Habermann and Notkin 1982] N. Habermann, D. Notkin "Gandalf Software Development Environment," in The Second Compendium of Gandalf Documantation, Department of Computer Science, Carnegie-Mellon University, May 1982.

[Pfleeger 1991] S.L Pfleeger*Software Engineering: The Production of Quality Software*, 2nd Edition, MacMillan, 1991.

[Ratcliff et al 1992] M. Ratcliff, C. Wang, R.J. Gautier, and B.R.Whittle "Dora - a structure oriented environment generator," *Software Engineering Journal*, Vol. 7, No. 3, 1992, pp. 184-190.

[Reiss 1985] S.P Reiss "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, Vol. 11, No. 3, 1985, pp. 276-285.

[Reiss 1990a] S.P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, 7, pp.57-66.

[Reiss 1990b] S.P. Reiss, "Interacting with the FIELD Environment," *Software - Practice and Experience*, Vol. 20, No. S1, 1990, pp. S1/89-S1/115.

[Stroustrup 1984] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass, 1984.

[StructSoft Inc 1992] *TurboCASE*, StructSoft Inc, 5416 156th Ave. S.E. Bellevue, WA 98006, 1992.

[Wang et al 1992] C. Wang, C-C. Leung, M. Ratcliffe, and F. Long, *Multiple Views of Software Development*, Technical Report, Computer Science Department, University College of Wales, Aberystwyth, 1992.

[Wasserman and Pircher 1987] A.I. Wasserman and P.A. Pircher "A Graphical, Extensible, Integrated Environment for Software Development," *SIGPLAN Notices*, Vol. 22, No. 1, 1987, pp. 131-142.

[Wasserman et al 1990] A.I. Wasserman, P.A. Pircher, and R.J. Muller, "The Object-oriented structured design notation for software design representation," *IEEE Computer*, Vol. 23, No. 3, March 1990, pp. 50-63.

[Welsh et al 1991] J. Welsh, B. Broom, and D. Kiong, "A design rationale for a language-based editor,"*Software - Practice and Experience*, Vol. 21, No. 9, 1991, pp. 923-948.