Integrated OO system development using SPE

John C. Grundy and John G. Hosking
Department of Computer Science
University of Auckland
Private Bag 92019
Auckland, New Zealand
Phone: (64) (9) 3737999 ext 8297
Email: john@cs.auckland.ac.nz

Abstract

SPE is a programing environment for OO programming 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 lifecycle providing support for conceptual level diagram construction, 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. This framework can be readily extended or tailored to support other languages and graphical notations.

1. Introduction

Diagrams illustrating relationships such as class inheritance, association and aggregation are an invaluable aid in understanding OO program structure [Coad and Yourdon 91, Wilson 90]. Diagram drawing can also be viewed as an act of program construction, either at the conceptual level during analysis and design [Henderson-Sellers and Edwards 90, Wasserman et al 90], as exemplified by CASE tools like the OOATool [Coad and Yourdon 91], or for creating compilable code, such as visual programming systems like Fabrik [Ingalls et al 88], Prograph [Cox et al 90], Garden [Reiss 86], and Ispel [Grundy and Hosking 91]. Diagrams are also useful when browsing programs [Fischer 87], visualising and debugging systems [Haarslev and Möller 90, Ingalls et al 88], and documenting and explaining systems at a high level of abstraction [Booch 91, Wilson 90].

However, diagrams alone are incapable of capturing all of the information required in a program. Low level code, such as method implementations, is more naturally and expressively programmed textually (although the dataflow approach of Prograph [Cox et al 90] attempts to show otherwise), and written documentation is always needed. Dora [Ratcliff et al 92], Mjølner [Magnusson et al 90], and PECAN [Reiss 85] are examples of systems which integrate textual and visual programming in one environment.

A challenge is to develop an OO programming environment which supports *all* of the various uses of diagrams, together with textual programming support, in one integrated programming environment. 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 programming environment in which high-level OO program structures are analysed, designed, constructed and browsed using graphical class diagrams. Code-level class interfaces and method implementations are programmed with free-edited text. Full consistency management between all phases of development is provided with design changes automatically modifying class and method interfaces and vice-versa. SPE also supports 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 in SPE. This is followed by a description of view construction mechanisms, view navigation, and program complexity management and browsing. Section 5 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, for example, shows some of the relationships between four classes of the drawing program. Information in a view may overlap with information in other views. For example, the `figure` class appears 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 5.

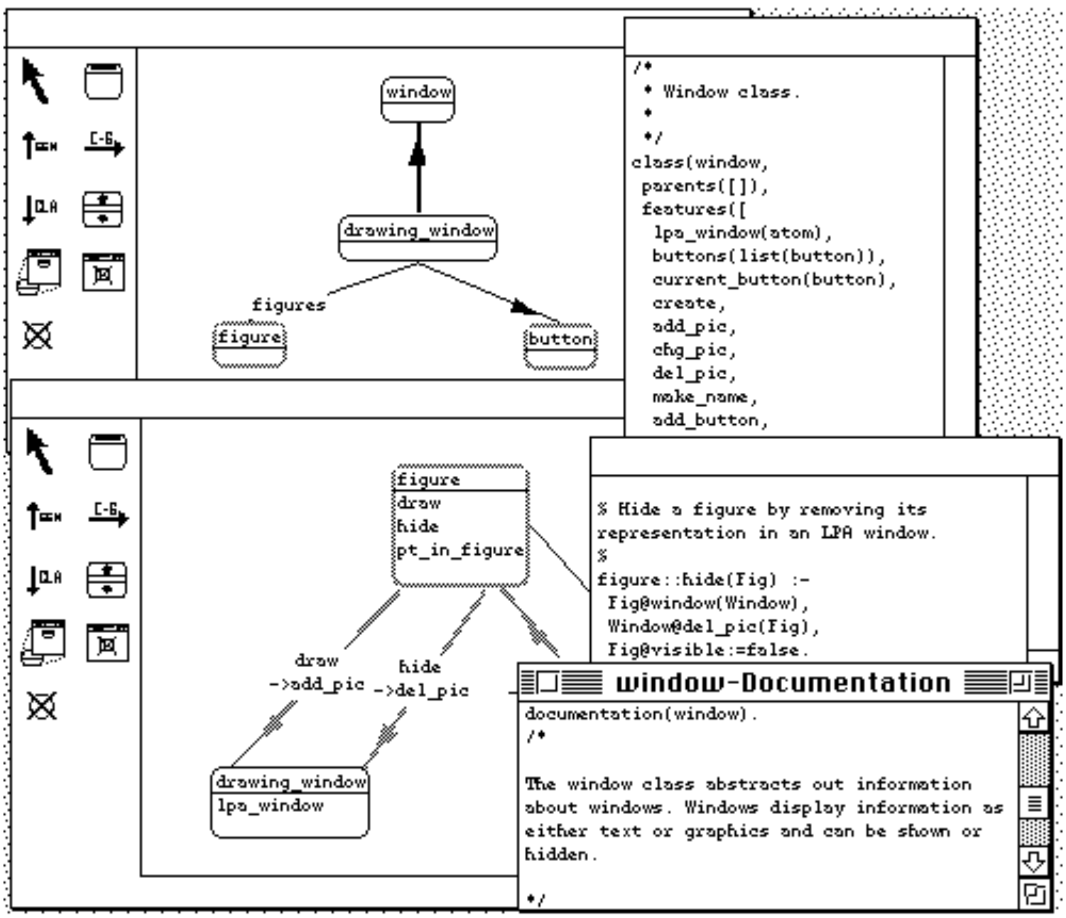

Fig. 1. An example SPE graphical and textual program views.

Fig. 1. also illustrates some of the elements 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 the grey icon border. Class icons can contain names of methods and attributes of the class, including those defined in the class or those inherited. Generalisation relationships are represented by bold arrows (e.g. between `drawing_window` and `window`). A variety of client-server relationships can be represented. These include aggregates (e.g. the `figures` feature of `drawing_window` holds a list of `figures`); feature calls (the `draw` method of `figure` calls the `add_pic` method of `drawing_window`); and abstract 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 syntax is that of Snart, an object-oriented extension to Prolog [Grundy and Hosking 92][1]. These different textual forms[2] can be mixed in one view or, as in Fig. 1, placed individually in separate views.

The ability to generate an arbitrary number of views combined with the 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.

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.

3. 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 an undo-redo facility allowing modifications to be undone. 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 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-server relationships. For the latter, additional information about the relationship (name, arity, whether it is inherited, and type - call, aggregate, abstract) is specified by dialog. Classes 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 server 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-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. SPE automatically lays out view extension information, but programmers are able to move these extra 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

---

[1] Snart takes a C++ like approach of separating class definitions and method definitions. Beyond that, the details of the language and its syntax are irrelevant for the purposes of this paper.

[2] 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.

"free-edit" mode of operation, in contrast to the "structure-edit" approach of the graphical tools[3], and then parsed to update base program information. This contrasts to most other environments, such as Dora [Ratcliff et al 92], Mjølner [Magnusson et al 90], and PECAN [Reiss 85] 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 by extension from other views and then modified in a free-edit fashion.

4. View Navigation and Browsing

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 90]). Click points allow rapid navigation to any other views (textual or graphical) containing the class or individual features of the class. SPE provides menus for textual views that perform similar facilities to click points.

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.

5. View Consistency and Update Records

Object-oriented software development tends to be an evolutionary process [Henderson-Sellers and Edwards 90, Coad and Yourdon 91]. Hence program design and implementation may require change for a variety of reasons:

- The requirements for the program change impacting on analysis, 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 91, Wasserman and Pircher 87] but few provide consistency management when code or design are changed.

5.1. Update Records

If a change to a system is made, for example a feature is renamed or its type changed, or a generalisation relationship is added to or removed from a class, this 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 it is recorded as an update record against the class and possibly some of its sub-components.

5.2. Propagation of Update Records

Update records are propagated from the view which is the source of the 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

---

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

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.

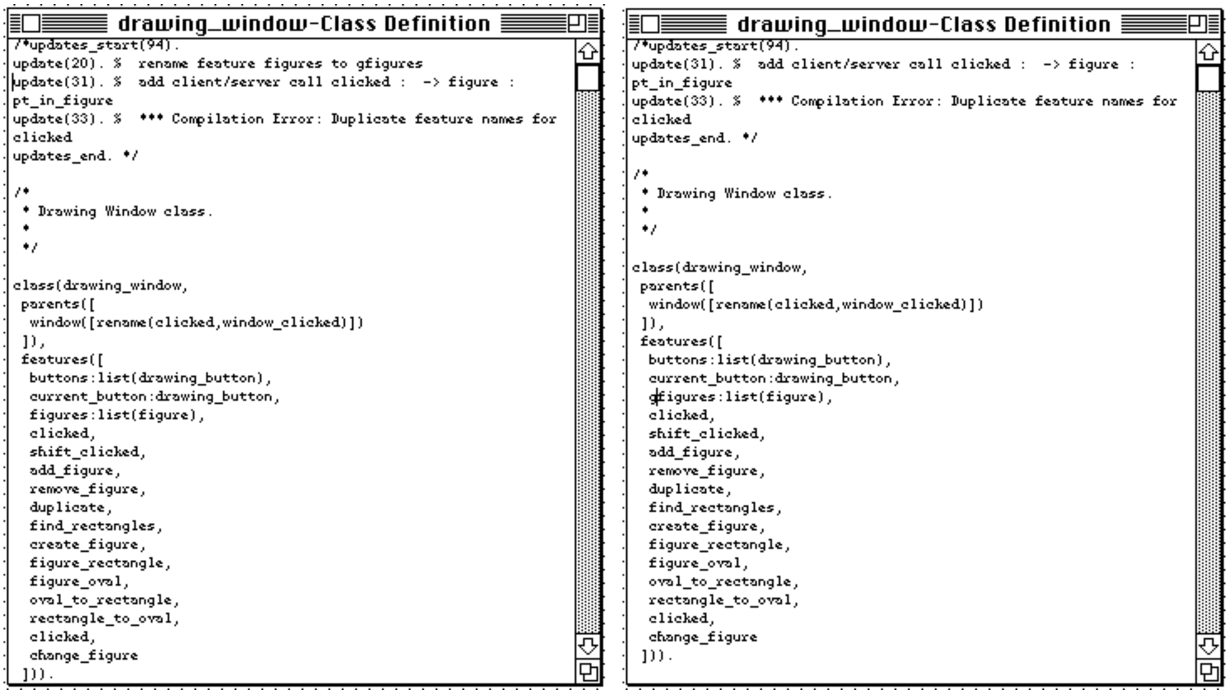

Fig. 2. (a) Updates expanded in a textual view, (b) first update applied.

Fig. 2 (a) shows a textual view with three update records added after corresponding changes in other views. The `updates_start` comment associated with all text forms is used as the position to place the update records. The first record is a change of the name of the *figures* feature to *gfigures*. Fig. 2(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. The second record in Fig. 2(a) is the addition of a client-server relationship indicating that method *clicked* calls method *pt_in_figure* of class *figure*. SPE can not automatically make a change for adding or removing this client-server connection because such a connection in a textual view is implemented as a feature call whose arguments and position in the method code can not be determined. The update is expanded, but the programmer is expected to make an appropriate change to the 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. The third record in Fig. 2 (a) shows such an example, where two features of the same name have been defined in a class.

For graphical views, updates from other 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 is removed), any inconsistent connection is drawn in colour to indicate the deletion.

5.3 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 (Fig. 3). 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.

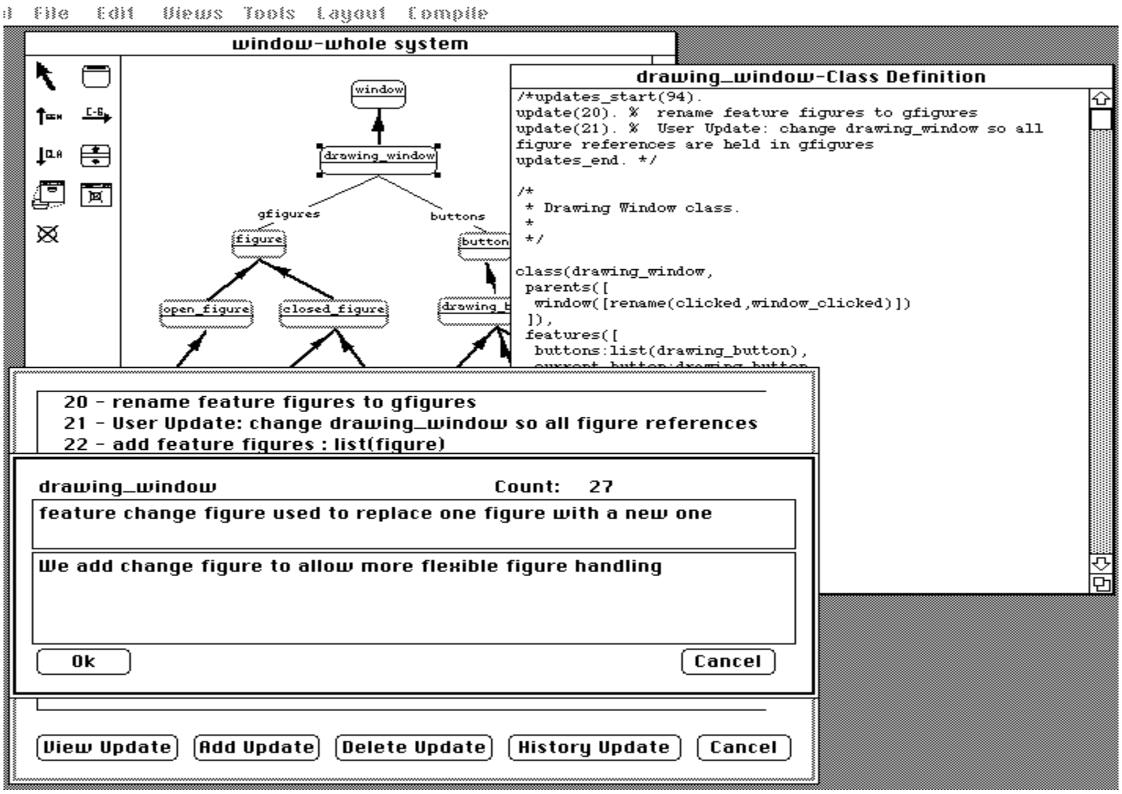

Fig. 3. Update viewing.

5.4 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 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 addition, 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 87]. 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.

6. Run-time Support

SPE uses the Snart compiler and run-time system to generate executable code and run a program. SPE provides dynamic visualisation views, called object views (Fig. 4). These interact with Snart's runtime system to display the state of run-time objects, including references to other objects, thus permitting the programmer to browse the execution state. Currently, the debugger of the Prolog system (LPA MacProlog) that Snart is implemented in is used to trace control flow between methods. However, we are working on extending SPE's object visualisation capabilities to include graphical object diagrams [Fenwick and Hosking 93] and provide data structure and control flow visualisation, similar to [Haarslev and Möller 90].

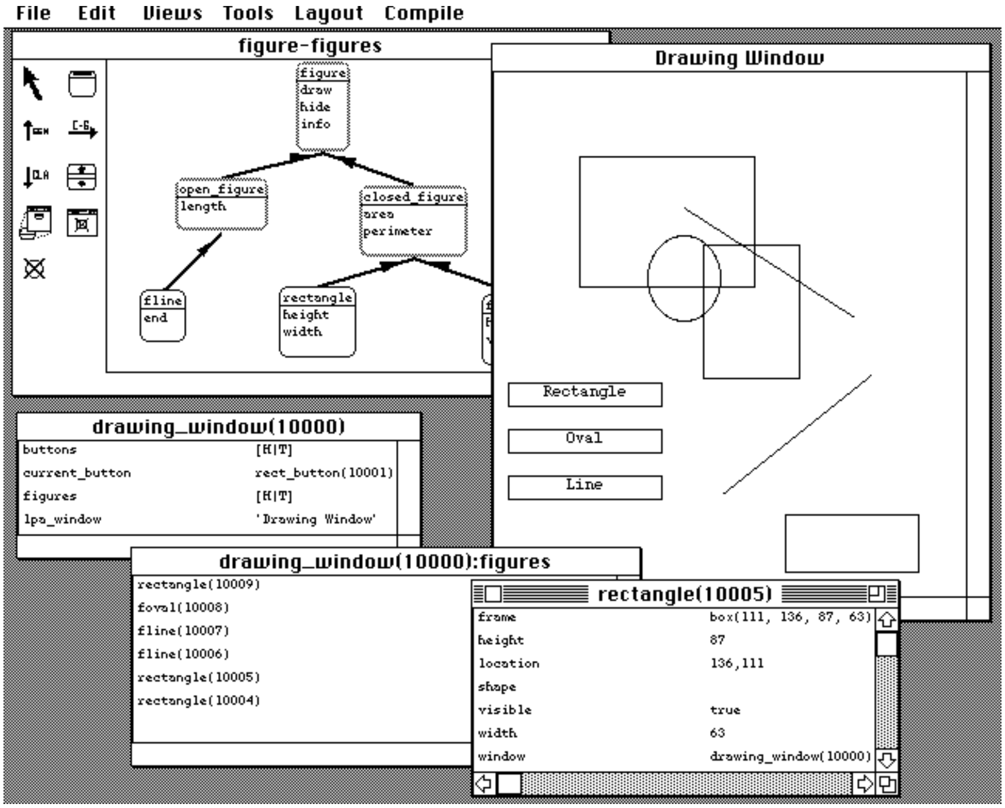

Fig. 4. Object views used in debugging the drawing program.

7. The MViews Framework

SPE is implemented as a specialisation of MViews, a generic framework for developing multiple-view based programming environments [Grundy and Hosking 92]. 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.

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 specializable 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. Also, generated PEs are well known for their poor user interfaces and performance [Minör 90]. 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.

Extension of IspelM to support other diagramming notations and OOLs is possible by 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 only several 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.

Modification to support other languages, such as C++ [Stroustrup 84], 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. These 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 92].

Compilation can be performed using the existing language compiler, either interfaced directly from the environment or via intermediate files. Some means of processing compiler errors (to convert them to appropriate update records) is also needed, possibly requiring development of a small purpose-built parser. To provide more immediate feedback on errors (particularly if the language compiler does not support incremental compilation) it may be desirable to encode some of the simpler semantic checking of the language, such as checking for multiple features of the same name, or inheritance cycles, within the environment.

Runtime support for Snart in SPE is implemented by direct interface with the Snart runtime system. For languages where this is impractical, other techniques must be used. Possibilities include automatic insertion of tracing and debugging code within the source (probably invisible to the programmer), as is done with Field [Reiss 90] or Zeus [Brown 91], or through introspection mechanisms available in dynamic languages such as Dylan [Apple Computer 92].

In each case the extensions to the IspelM framework are straightforward, and involve no technical issues that have not been solved previously. The major effort is likely to be in generating the parsers and unparsers for textual views, and "glue code" for communicating results between the environment, the compiler, and the runtime system.

8. 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 lifecycle. 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 update record mechanism, which also provides 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 (moderated by update records), version control and macro-editing operations using update records, and at improving execution time support facilities, based on preliminary work by [Fenwick and Hosking 93].

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 93].

Acknowledgements

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 work.

References

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

Apple Computer 1992: Dylan an object-oriented dynamic language, Apple Computer Inc, Cupertino.

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

Brown, M. H., 1991: Zeus: A system for algorithm animation and multi-view editing, Proc IEEE Workshop on Visual languages, 4-9.

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, Proceedings of 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, Technology of Object-Oriented Languages and Systems TOOLS 5 Proc of the 5th International Conference Santa Barbara, Prentice Hall, 129-138.

Grundy, J.C., Hosking, J.G. 1992: MViews: A Framework for Developing Visual Programming Environments, Technology of Object-Oriented Languages and Systems TOOLS 9 Proc of the 9th International Conference Sydney, Prentice Hall, 129-137.

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.

Meyer, B., 1992: Eiffel the language, Prentice Hall, Herts, UK.

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

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.

Reiss, S.P, 1990: Connecting tools using message passing in the Field environment, IEEE Software, July, 57-66.

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

Stroustrup, B., 1984: The C++ Programming Language, Reading, Mass, Addison-Wesley.

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.

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