A Visual Programming Environment for Object-Oriented Languages

J.C. Grundy, J.G. Hosking, J. Hamer
Department of Computer Science
University of Auckland,
Auckland, New Zealand
jgru1@cs.aukuni.ac.nz

Abstract

Ispel is a visual programming environment for object-oriented languages providing multiple views of programs, utilising both graphics and text. These views can be used not only for program browsing, but can also be manipulated to visually program with an object-oriented language.

1. Introduction

In the course of developing several large object-oriented programs, the authors have found diagrams representing inter-class relationships to be a useful design and documentation aid. The usefulness of these diagrams suggested that they could form the basis of a visual approach to programming with object-oriented languages. The principal aim of the work presented here is to improve the environments for two object-oriented languages, *Kea*[1] (Hosking *et al*, 90, Mugridge *et al*, 91) and *Eiffel* (Meyer, 88), by extending the diagramming techniques into a visual programming system. The two languages are quite different in detail, but share a common approach to strong typing, information hiding, and polymorphism. Current environments for both languages are quite rudimentary, with few support tools, and those that exist being poorly integrated.

We begin by examining class structure diagrams and their uses. We then concentrate on their use for visual programming and develop a set of criteria for a visual programming tool. Primary amongst these criteria is the need to allow multiple views of a program, with shared information, and consistency under change. The remainder of the paper describes Ispel, a visual programming tool supporting the design criteria.

2. Class Structure Diagrams

The classes, features, and relationships that comprise an object-oriented program can be naturally and clearly expressed using diagrams (Booch, 86; Meyer 88; Wasserman *et al*, 90; Wilson, 90). Such *class structure diagrams* can represent relationships such as generalisation (inheritance) and aggregation (feature hierarchies). Class structure diagrams, as used here[2], are composed of boxes and lines laid out to form a meaningful representation of an object-oriented program. An alternate approach is the nested box approach of the Mjölner environment (Hedlin and Magnusson, 88).

Fig. 1 is a class structure diagram for a *Kea* program called Wallbrace, showing a simple feature hierarchy. Wallbrace assists a building designer to check conformance of a building with the wall bracing requirements of a code of practice for timber frame houses (Mugridge and Hosking, 88). Fig. 2 is a class structure diagram showing inheritance relationships between some of the Wallbrace classes.

[1] A kea is a colourful New Zealand alpine bird. *Kea*, an object-oriented functional language, was formerly know as *Class*.

[2] There are many notations in the literature. This is one the authors have found useful.

The arrows represent a class inheriting from its parent. Diagrams mixing both aggregation and inheritance are also useful.

Class structure diagrams are useful in several areas of object-oriented programming (Wilson, 90):

- *Analysis and Design*. Diagrams present the structure of a program for programmers to understand (Coad and Yourdon, 91). They can assist in choosing classes, features, and generalisations, and aid program structuring (Wasserman *et al*, 90). Tools to support this include: OOATool™ (Coad and Yourdon, 91); Software through Pictures (Wasserman and Pircher, 87); and Graspin (Mannucci *et al*, 89).

- *Documentation and Browsing*. Diagrams are useful in presenting a finished design to others to help them understand or maintain programs. Diagrams (particularly inheritance ones) can often be automatically derived from program text and may then be used for documentation, or as the basis for a browsing tool. Examples of this include the **good** browser for *Eiffel* (Interactive, 89), the browsers in the Trellis/Owl (O'Brien *et al*, 87) and ObjTalk environments (Fischer, 87), and the THINK Pascal environment (Symantec, 89).

- *Implementation*. Diagrams can form the basis of implementations. In *visual programming* construction of a diagram causes construction of all or part of an executable program (either with or without an intermediate textual representation). Systems which support object-oriented visual programming include: Prograph (Gunakara, 89); Fabrik (Ingalls *et al*, 88); PECAN (Reiss, 85); Garden (Reiss, 87); and the Mjölner environment (Hedlin and Magnusson, 88).

- *Debugging*. Diagrams can be used to describe the execution state of an object oriented program, such as in the Mjölner environment (Hedlin and Magnusson, 88). Related to this are the areas of program visualisation and algorithm animation (Ambler and Burnett, 89; Myers, 90).

These uses tend, however, to be distinct with little overlap. As can be seen, diagrams are useful in all phases of design and implementation, and thus integrated support throughout the design-cycle is needed. As a first step to achieving an integrated environment, we have focussed on a visual programming tool supporting both diagrammatic design and implementation plus browsing and documentation. Given the indistinct boundary between object-oriented analysis and design (Coad and Yourdon, 91), we feel this tool will be useful for analysis too.

3. Design Criteria

Following our experience with class structure diagrams and observation of other systems, the following design criteria for a diagramming tool to support design and implementation of *Kea* and *Eiffel* programs were established:

- A specialised package is needed for drawing diagrams during analysis and design. This is essential for productivity (Coad and Yourdon, 91). Drawing class structure diagrams by hand or using a standard drawing package is tedious.

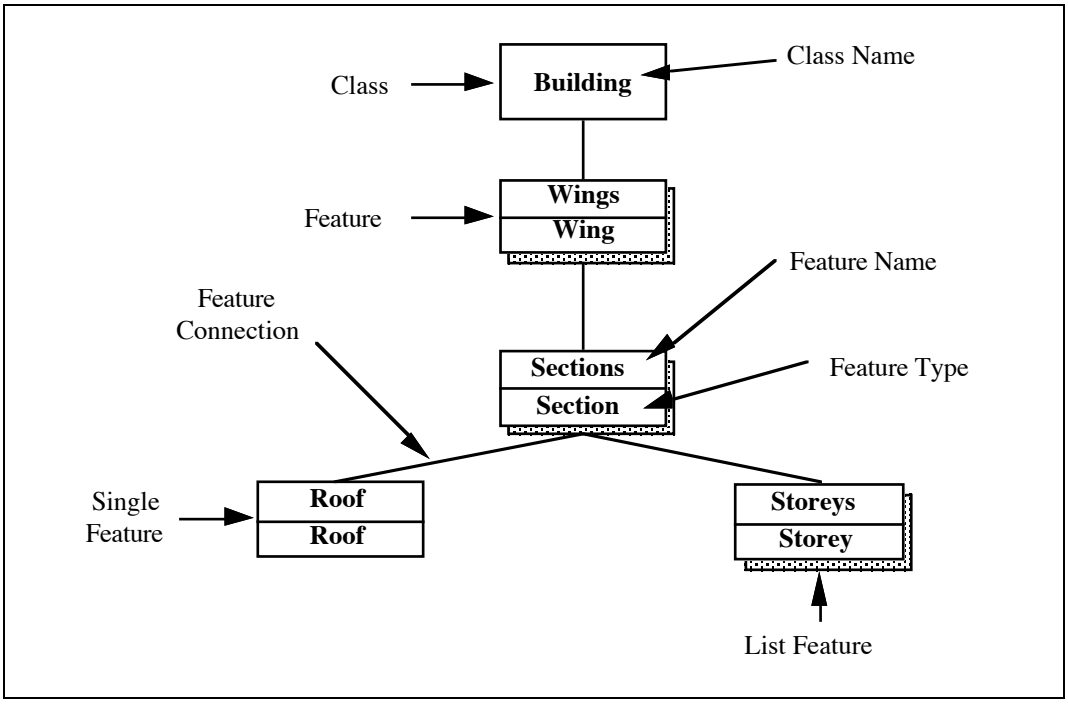

Figure 1. A class structure diagram from the Wallbrace system.

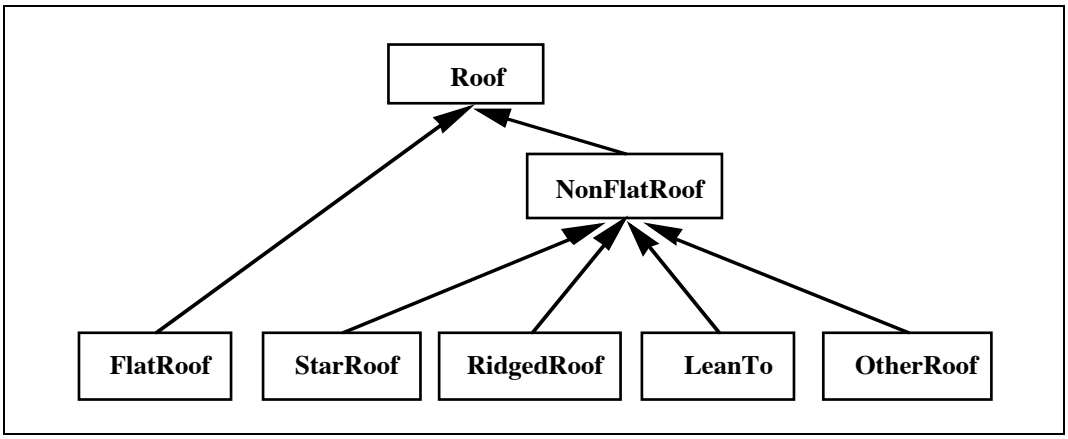

Figure 2. A class structure diagram for generalisation.

- Diagrams should be automatically translated to the implementation language, i.e visual programming is supported. Preferably this should be done incrementally so that errors can be trapped as the diagrams are constructed.

- Diagrams are not appropriate for everything. The authors' view is that text is more appropriate than diagrams at the detailed expression level, while diagrams are more useful for the high level object-oriented structuring of an application. The point at which one becomes more useful than another varies both at different places in the design cycle and between different programmers. Hence the ability to choose the appropriate representation to work in at any stage is important and it is essential that the diagrams for a program be integrated with the program text. Thus separate views for each representation should be available and the programmer should be free to move between them.

- Multiple diagrams with shared information are essential. We have found that as applications become complex, the ability to construct partial views of the application is essential. Simple examples of this are separate

views for inheritance graphs and feature hierarchies. The WallBrace system provides a more complex example. In this application, the processing consist of three conceptual (but interleaved) phases: entry of plan information, loadings calculation, and bracing requirement calculation. These phases are pervasive, each having effects on most of the classes in the application. Accordingly it was found useful to construct separate class structure diagrams for each of the phases, each being a "vertical slice" of the complete system ignoring features, subclasses, and other detail concerning only the other phases. However, these diagrams are by no means independent, sharing many common classes, features, inheritance links, etc. The application as a whole is then a union of the information in each of the diagrams.

- With multiple diagrammatic and textual views available, automatic consistency management becomes essential. Modification of information in one of the views (diagram or text) should be propagated in an appropriate way to other views affected by the modification. This allows programmers to move between representations in a consistent manner.

- Diagrams should not be "static". Having constructed diagrams for the purposes of designing and implementing a program, they can be used dynamically to view and navigate through a developing application, forming the basis of a browsing tool, and as "active" documentation.

The most interesting aspects of this specification are the multiple diagrammatic and textual views, particularly the emphasis on "vertical slices", and the consistency management problem. Many systems integrate textual and graphical views, e.g. PECAN (Reiss, 85) and the Mjölner environment (Hedlin and Magnusson, 1988) but few provide the free interchange between representations proposed here, nor the "vertical slice" approach to multiple diagrams. An exception is the Garden system (Reiss, 87) which appears to provide both of these capabilities, but with somewhat less emphasis on the latter. Consistency across views is usually provided, as in PECAN, by allowing modification from only one view and one-way propagation to other views. Only the Garden system appears to provide an inter-view consistency manager of the sort advocated here, although Reiss (87) acknowledges that formally specifying the semantics of consistency checking is extremely difficult.

We have designed and prototyped a visual programming environment for *Kea* and *Eiffel*, called Ispel, following the above design criteria. The basic concepts of the Ispel design, and use of the environment when developing a program are described in the following section.

4. Ispel

Ispel uses the desktop metaphor. Windows contain the diagrams (views) for a program; menus and dialogues are used for user interaction. Boxes and lines comprising a class structure diagram are manipulated using a tool palette and mouse. Additional programming tools are integrated into the environment using the same interface. Fig. 3 is a screen dump of a Macintosh prototype of Ispel.

4.1. Views and Visual Programming

Ispel allows multiple class structure diagrams (views) for a program. Views contain boxes and lines, representing classes, features, and feature or inheritance relationships. Each view represents a particular focus on some aspect of a program. Views are focused around one class: the *primary class* for the view. For example, *Building* in Fig. 3 is the primary class for that view.

Views can share information, so a class or feature can appear in more than one view. Classes can also be the primary class for more than one view. This allows a programmer to construct views for different aspects of a program, and to

represent different information in each view. Views can be displayed in different windows, so more than one view is visible at a time. Fig. 4 shows three views from Wallbrace.

Programmers decide when a new view or window is to be created, and what classes and features are included in the view. A class can be expanded to show existing features, subclasses, generalisations, etc. Fig. 5 shows expansion of the *FlatRoof* class of Wallbrace. Its generalisation (*Roof*) and two levels of features are displayed. The expanded details are given a default layout, but Ispel then allows a user to move boxes around, the lines connecting the boxes being automatically redrawn. Boxes can be hidden (no program modification) from views, in which case the boxes and lines below the box are also removed from the view.

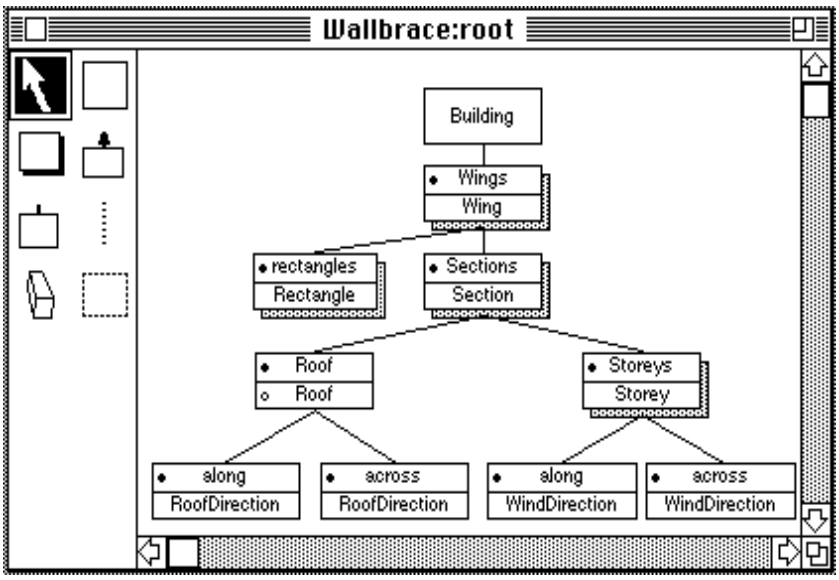

Figure 3. Screen dump of Ispel showing part of the Wallbrace system.

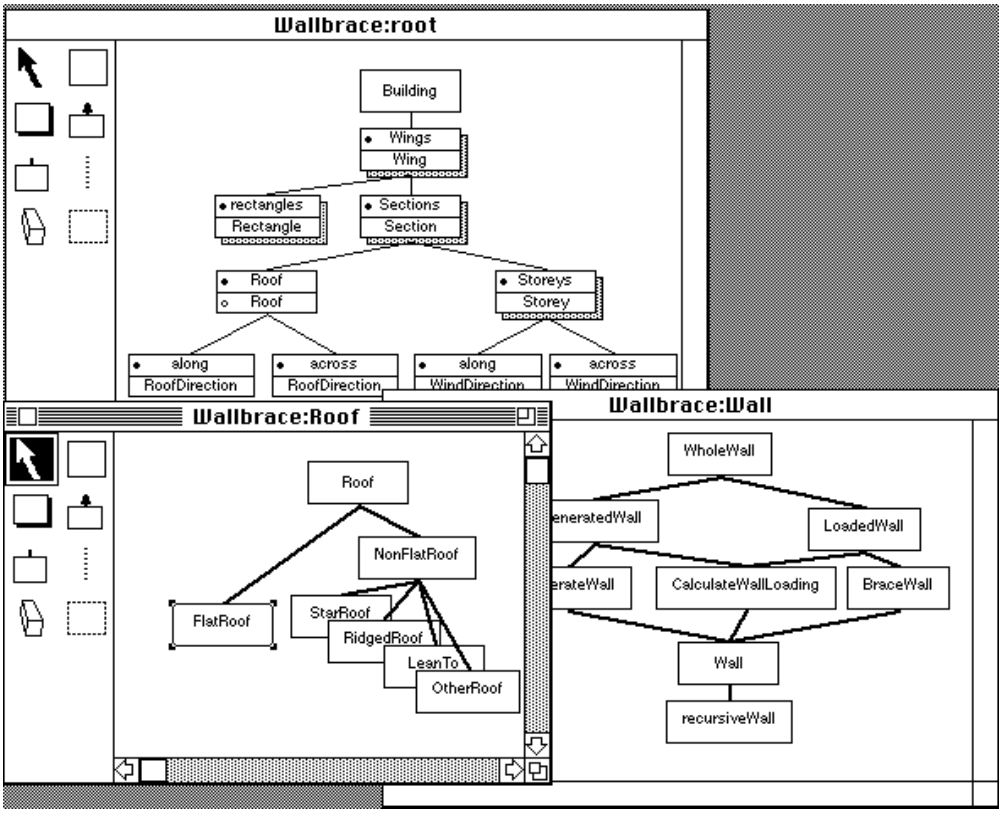

*Figure 4. The **Building**, **Roof**, and **Wall** views from Wallbrace.*

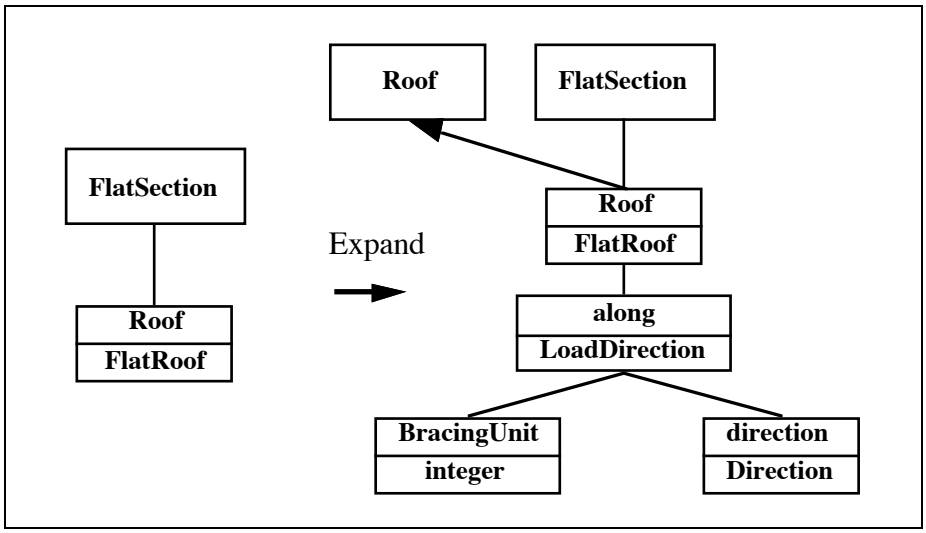

Figure 5. A class being expanded and automatically laid out by Ispel.

Boxes and lines representing classes, features, and inter-class relationships can be *added* to views (causing program modification) using the palette provided. Lines can be connected to the top and bottom of boxes, or to the sides of boxes. Boxes and lines can be cut from a view, resulting in modification of the program, and other views possibly being updated. Classes and features can be renamed, and one class can be selected to replace another in a diagram. More detailed information can optionally be represented in a graphical view. This includes the visibility of features (public or private), and whether a feature is a class parameter, procedure, or function.

Programmers can create new views for a class, initially containing only the chosen class. Additional information is added by expanding existing boxes or adding new ones. Additional windows can be created allowing several simultaneous views. Multiple programs can be constructed at one time, and both the graphical and

textual aspects of programs can be saved to files.

4.2. Visual to Textual Cross-Over

Not all programming is performed using graphics in Ispel. High-level, object-oriented aspects of programs are constructed and viewed visually, but implementation of feature bodies is coded in text. We feel that the expression level aspects of *Eiffel* and *Kea* are better suited to textual construction. Editing of the textual representation of classes is integrated with the visual programming. A textual representation of a class can be obtained, by double-clicking on the class box, and then edited. Fig. 6 shows a graphical and textual representation of the *Roof* class from Wallbrace.

The current Ispel implementation only supports one textual view of a class, showing all of the contents of that class, not just those displayed in the graphical view the textual view was accessed from. Future implementations will include multiple textual views, allowing textual views of a class to be tailored in a similar manner to the diagrammatic views.

4.3 Consistency

An important aspect of diagrammatic views is that they are always kept consistent. If several views share information, and that information is changed in one view, all views are updated to reflect the change. For example, if the feature *Roof* is deleted from class *Section* in Fig. 4, all views including this feature are updated to reflect the change.

Cross representational consistency between diagram and textual views is not as well developed as yet. Propagation of changes is currently only unidirectional from diagram to text. For example, if the feature *across* is deleted from the graphical view in Fig. 6, then the textual view is updated appropriately. Consistency between the graphical and textual representations is maintained by using a common underlying representation for views. Propagation of changes in textual views to corresponding graphical views is currently being implemented.

When complete, this approach will have the advantage that all diagram and textual representations of programs are consistent with each other, with consistency maintained by the environment, not the programmer. The integration of graphics and text provides flexibility in that a programmer can choose the appropriate representation in which to construct programs.

4.4. Program Navigation Using Views

Multiple views can be used to browse through a program. The structure of the program can be viewed in many different ways, and additional views created to aid program navigation. As the textual representation for the implementation of classes can be accessed via the graphical views, Ispel provides a high-level structure for accessing implementation details.

To navigate between views, two complementary facilities are provided. Classes have a view icon which, when double-clicked, changes the view. Each class has one view called a primary view,

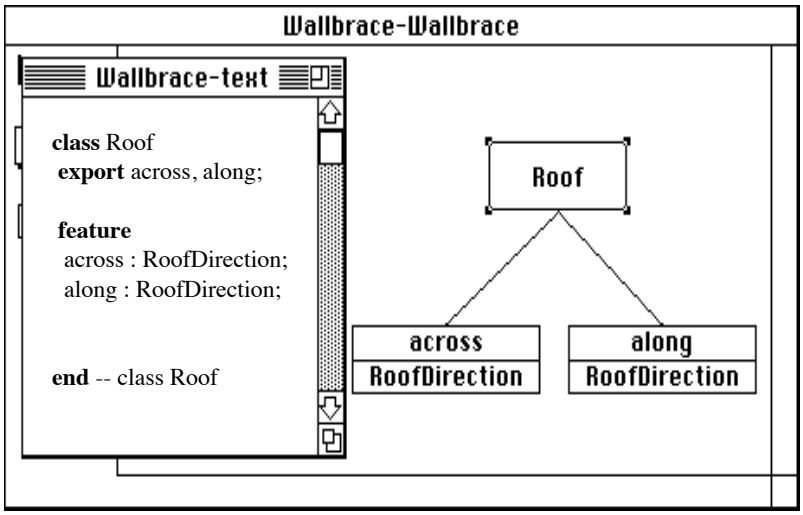

*Figure 6. Visual and textual views of the **Roof** class in Wallbrace.*

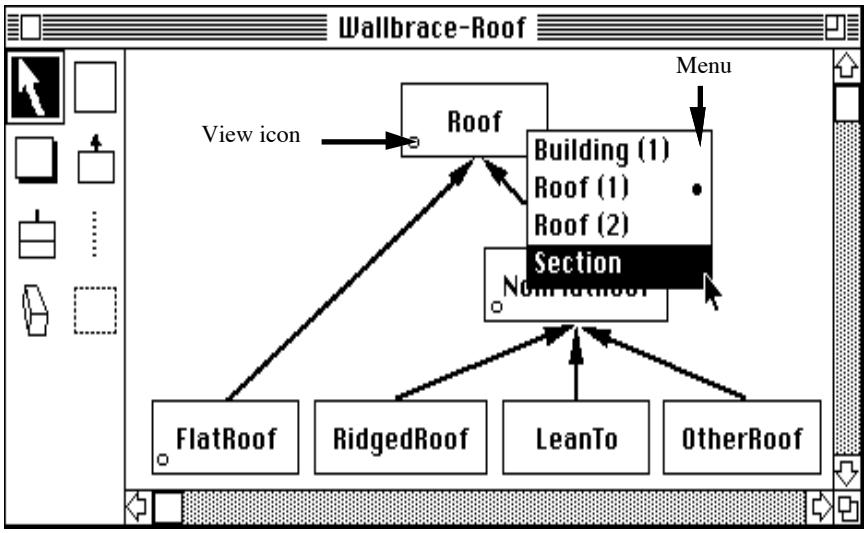

Figure 7. An example of view navigation in Ispel.

which is the "most useful" view for the class. Double-clicking on the class view icon selects this view. The programmer can change the primary view for a class if required to assist view navigation. As classes can have several views for which they are the primary class, and can also be used in many other views, a second facility provides menus for view selection. Fig. 7 shows an example of view navigation. *Roof* occurs in the view for the *Section* class, which the programmer is about to display. Views can be displayed in different windows or the same window (to avoid screen clutter).

5. Structure of Ispel

Ispel uses a three level architecture, shown in Fig. 8. The top level is the screen representation of both the visual and textual versions of the program, the level that a user of Ispel sees and manipulates. The middle level contains abstract versions of the visual and textual representations, eliminating layout information. The visual representation is composed of views, i.e class structure diagrams. The textual representation is a collection of classes including information, such as expressions, not represented visually. The bottom level is an abstract representation of the object-oriented parts of the entire program. The main purpose of this layer is to provide

consistency between each of the views and also with the textual representation. Changes to this layer are propagated to views and the program text.

The purpose of each layer is clearer if we examine the sorts of operations that affect each layer. The table below gives examples. The important point to note is that if a change is made to information in the common layer, that change must propagate to each view (diagram or text) using that information.

A formal model of Ispel's architecture has been developed which defines the visual and common levels in terms of graphs and operations that manipulate these graphs. Formalising Ispel has had several benefits. The way in which different aspects interact is fully defined, and consistency between different representations maintained. As operations that can change Ispel's state are defined formally, they can be proven correct, and thus the environment proven to work as it should. Formal definition also gives an abstract view of the environment, serving as a specification for how the basic aspects of Ispel should behave, and how they must be kept consistent.

Changes	**Visual**	**Textual**
Screen only	Move a box	Add white space
Screen and View	Hide a box Expand a box	Modify a feature body
Screen, View and Common OO	Add a feature box Remove a feature box	Add a feature Remove a feature

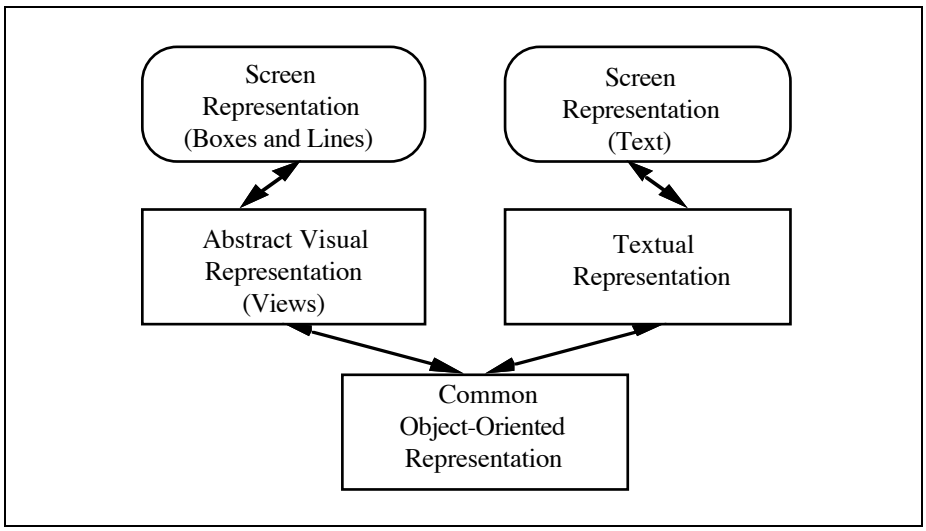

Figure 8. Representation levels in Ispel.

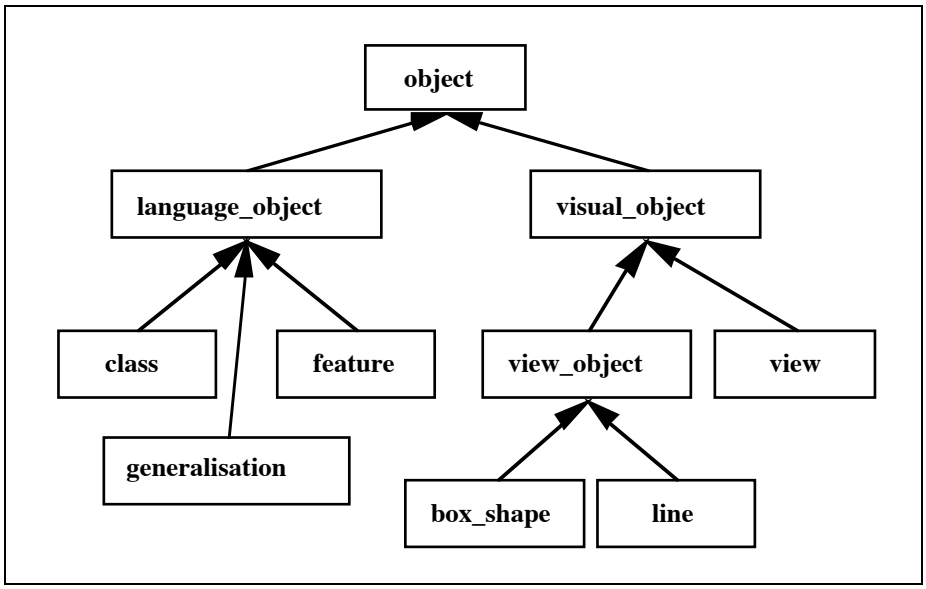

Figure 9. Classes representing language elements in Ispel.

We have developed two prototypes of Ispel and are in the process of implementing a more complete development environment. The first prototype of Ispel was developed using LPA MacProlog™ on the Macintosh. The purposes of this prototype were to determine if visual programming was applicable for object-oriented languages, and to refine the user interface and visual programming aspects of Ispel. A second prototype of Ispel was implemented using *Eiffel*. The purpose of the *Eiffel* prototype was to develop and refine an object-oriented implementation of Ispel, and to identify important parts of the formal definition.

Many aspects of a visual programming system are appropriately expressed in an object-oriented manner, e.g. the objects that comprise a system share attributes and functions at different levels of abstraction. Fig. 9 shows the hierarchy representing the visual and textual language elements of a *Kea* program within Ispel. Ispel operations are also expressed as objects, and genericity and generalisation permit reuse, polymorphic assignment, and code sharing between operations. Expressing operations as objects also allowed an *undo* facility, important for interactive software (Reiss, 85), to be provided at minimal cost. Relationships, such as inheritance and feature, are also expressed using objects, as is the framework for the environment. This model of implementation is similar to the OROS (Object, Relationship, and Operation System) model of the Arcadia environment (Rosenblatt *et al*, 89). Key ideas of inter-object dependency, propagation of changes via relationships, and visual representation of an underlying program have been developed as part of this model.

6. Conclusions and Future Research

We have described Ispel, a visual programming environment for object-oriented languages. We have found Ispel to be useful for programming object-oriented systems. Most of the diagrams in this paper have been drawn using the Prolog prototype of Ispel. Reconstruction of parts of the Wallbrace system with Ispel has demonstrated that this environment significantly enhances the speed of program development. The object-oriented design of the *Eiffel* prototype was constructed and modified using the Prolog prototype.

The major conclusions of our work are:

- A multiple view "vertical slice" approach to both visual programming and program browsing allows programmers to focus on particular aspects of a program. In the process, other irrelevant information can be excluded. Using the set-of-services

approach to class construction advocated by Meyer (1988) leads to classes containing many features, but where these features tend to be naturally grouped into common service areas. Constructing multiple views, one concentrating on each group, allows the complexity of such classes to be managed.

- Consistency between views is critical. Our formal model is one step towards providing a semantics for consistency management, while the object-oriented implementation provides a base set of tools to implement those semantics.

Currently, we are developing a full implementation of Ispel using C++ with an X windows graphical user interface. This will provide an environment intended for multi-user development, with interfaces to the C++, *Eiffel*, and *Kea* compilers and run-time systems. It will help to further refine many concepts of Ispel, and assist enhancement of the implementation and formal models. A performance evaluation of the environment is planned to determine how much assistance such a tool gives to object-oriented development.

The diagramming notation for Ispel will be extended to provide more visual programming power, and include more powerful design and implementation facilities. We have noted that there are many commonalities between aspects of direct manipulation, diagramming, and visual programming systems. We hope to simplify the construction and modification of all of these systems by abstracting the implementation and formal models for Ispel so that more generic environment components can be produced for reuse.

Acknowledgements

The authors would like to thank Iain Shearer and Rick Mugridge for their help in developing this research project. The financial assistance of Mana Systems Ltd, the Building Research Association of New Zealand, and the University of Auckland Research Committee is gratefully acknowledged.

References

Ambler, A., Burnett, M., 1989: "Influence of Visual Technology on the Evolution of Language Environments", IEEE Computer, October, pp. 9-22.

Booch, G., 1986: "Object-Oriented Development", IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February, pp. 211-221.

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

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

Gunakara Sun Systems, 1989: "Prograph Reference Manual", The Gunakara Sun Systems Ltd.

Hedlin G., Magnusson, B.,1988: "The Mjölner environment: direct interaction with abstractions", Proc ECOOP '88, pp41-54.

Hosking, J.G., Hamer, J., Mugridge W.B., 1990: "Integrating functional and object-oriented programming", Proc TOOLS Pacific, Sydney, pp. 345-355.

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

Interactive, 1989 "Eiffel: The Environment", Technical Report TR-EI-5/UM (Version 2.2), Interactive Software Engineering Inc., August.

Mannucci, S., Mojana, B., Navazo, M.C., Romano, V., Terzi, M.C., Torrigiani, P., 1989: "Graspin: A Structured Development Environment for Analysis and Design", IEEE Software, November, pp. 35-43.

Meyer, B., 1988: "Object-Oriented Software Construction", Prentice-Hall.

Mugridge W.B., Hosking J.G., 1988: "The development of an expert system for wall bracing", Proc 3rd New Zealand Expert System Conference, Wellington, pp. 10-27.

Mugridge W.B., Hamer, J., Hosking J.G., 1991: "Multi-methods in a statically-typed programming language", to be presented to ECOOP '91, Geneva, July.

Myers, B.A., 1990: "Taxonomies of Visual Programming and Program

Visualization", Journal of Visual Languages and Computing, Vol. 1, No. 1, March, pp. 97-123.

O'Brien, P.D., Halbert, D.C., Kilian, M.F., 1987: "The Trellis Programming Environment", OOPSLA '87 Proceedings, pp. 91-102.

Reiss, S.P., 1985: "PECAN: Program Development Systems that Support Multiple Views", IEEE Transactions on Software Engineering, Vol. 11 (3), March, pp. 276-285.

Reiss, S.P., 1987: "Working in the GARDEN Environment for Conceptual Programming", IEEE Software, November, pp. 16-26.

Rosenblatt, W.R., Wileden, J.C., Wolf, A.L., 1989: "OROS: Toward a Type Model for Software Development Environments", OOPSLA '89 Proceedings, pp. 297-304.

Symantec Corporation, 1989: "THINK Pascal Reference Manual", Symantec Corporation.

Wasserman, A.I., Pircher, P.A., 1987: "A Graphical, Extensible, Integrated Environment for Software Development", SigPlan Notices, Vol. 22, No. 1, January, pp. 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, pp. 50-63.

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