Direct manipulation of an object store

by Pete Sawyer and Prof. Ian Sommerville

Integrated project support environments (IPSEs) are intended to provide a cohesive and integrated set of tools to support the process of design and development in software engineering projects. Much current research is concentrated on maximising the degree to which these tools can be integrated. This paper briefly describes the architecture of a prototype IPSE which attempts to achieve a high degree of integration using techniques drawn from the disciplines of intelligent knowledge-based systems, office automation and object-oriented programming. The remainder of the paper deals with the design of a user interface to the IPSE based on direct manipulation. It argues that this provides a consistent and integrated method with which users can interact with the objects in the IPSE's object store.

1 Introduction

The current generation of integrated project support environments (IPSEs) provides tool kits designed to aid the construction of large software engineering systems. The breadth of support and degree of integration which these IPSEs achieve are generally greater than that provided by the earlier programming environments from which they are linearly descended. Partly as a result of their ancestry, however, they fall some way short of providing completely *integrated* support because they embody the notion that the tools should be applied to project components by users.

It has been suggested that intelligent knowledge-based systems techniques have the potential to give software engineering tools a degree of built-in intelligence which would reduce their dependence on user-initiated actions. For this idea to be successful, these smart tools would need to be part of an IPSE structure which itself possessed a degree of knowledge about the components residing within it. The IPSE would need to be able to view a project as a cohesive whole, instead of each component in isolation. It is clear that if an IPSE capable of achieving these goals is to be built it will necessitate a radical departure from the design of existing IPSEs.

The ISM project is intended to demonstrate the potential of the application of artificial intelligence (AI) techniques to the support of software engineering projects. The project's resources have been concentrated on designing a set of basic facilities for a knowledge-based IPSE and producing demonstration applications of support for some of the high-level activities involved in software engineering. These have historically received less attention than lower-level, programming-oriented activities which many existing environments support.

The ISM knowledge base includes a model of each project which it supports. This project model encapsulates the knowledge which, early in the history of programming environments (Ref. 1), was identified as being necessary for tackling the problem of complexity in large software engineering projects. Information about a project, its aims, products, resources, staff etc. is embodied within ISM as objects which encapsulate the system's knowledge about them. ISM is able to reason about an evolving project and take over some tasks which, in current IPSEs, are the responsibility of humans.

If ISM is to provide truly integrated support for software development, it is essential that users are presented with a consistent way of viewing the state of the system. A user interface based on the concept of direct manipulation of ISM objects is being designed. It is argued that this presents the user with an easy-to-use interaction style and provides a natural mapping on to the underlying data representation.

2 Background

In common with any engineering task, the activity of software engineering can be aided by the use of appropriate tools. Much effort has been invested in producing software engineering tools: compilers, symbolic debuggers etc. Programming environments arose from the idea of collecting tools together into sets where they could be invoked as appropriate during the course of software development.

UNIX (Ref. 2) is a successful example of such a programming environment. Here the sets of tools provided by the various shells are superimposed on the underlying operating system kernel; components associated with software development (source code, object modules, documents etc.) are simply held in the filestore. In programming environments like UNIX, the user (developer, programmer) has full responsibility for the application of the correct tools to the appropriate components of the design exercise. The environment itself encapsulates little information regarding interdependencies of tools and components. For this reason, a programming environment such as UNIX cannot be regarded as providing integrated support; rather, it is a collection of independent, manually invoked tools.

A project support environment differs from a programming environment by providing support for all the activities involved in a project, from initial specification through to product maintenance. Only if that support is provided in such a way that the environment views the various tools, not in isolation, but as a set of interdependent activities, can it be said to be an *integrated* project support environment.

Aspect (Ref. 3) and Eclipse (Ref. 4) are typical of current IPSE designs in which a tool set is layered around a project database, access to which is governed by an object management system. The object management system provides mechanisms for maintaining consistency among project components to a greater degree than filestore-based environments are capable of.

The current generation of IPSEs embodies a fundamental limitation to the degree of integration which they achieve because they are essentially passive. That is, they exhibit integration on two levels:

- data integration via a database management system
- user interface integration via a consistent metaphor and standards.

They do not support activity integration in as much as the activities involved in the software process are initiated entirely by user actions.

ISM is an attempt to address the problem of activity integration. It uses knowledge-based systems techniques to tackle the relatively unstructured nature of the software process and the non-deterministic patterns of activity activation.

The advantages of an active IPSE with activity activation are as follows:

- It is possible to use a model of the development process to drive that process. Development process activities may be associated with IPSE agents which are activated asynchronously by the presence or absence of some data. Thus process quality and hence product quality are improved because some of the uncertainties and informality of human process management are removed.
- Software productivity is increased because developers are given more 'intelligent' assistance by the environment. They need not set out all actions in detail, but may simply specify a goal which the active environment should attain.
- Resource management is improved because the environment can maintain a detailed map of available resources and schedule these accordingly. Incomplete user knowledge of resources is often a major constraint on development.

The ISM environment replaces the concept of tools by that of *agents*. In contrast to current environments where tools are layered around the IPSE kernel, ISM consists of a federation of agents which embody sufficient contextual knowledge to be invoked on an opportunistic basis. ISM is able to reason about an evolving project, integrate the various transformations which need to be applied to components, and automate many software process tasks. The ability to reason about a project and generate transformations automatically embodies the IPSE with the attribute of being *active*.

ISM's active attributes enable the environment to assume responsibility for much of what has hitherto been part of the IPSE user's workload. An example of this is provided by an intelligent planning agent that would automatically generate alternative plans for re-scheduling staff and resources in the event of a project milestone failing to be delivered on time. Bug fixes would be automatically distributed to recipients of affected system versions by an intelligent configuration management agent.

In both Aspect and Eclipse attempts have been made to provide user interface dialogue styles which are easier to learn and more informative than the command languages employed by earlier environments. The Eclipse user interface employs a *control panel metaphor* (Ref. 5). Users interact with tools by using a pointing device to select graphical images on their workstations' screens as if they were pressing buttons, setting toggles etc. on a piece of hardware. A *user interface design tool* (Ref. 6) extends the metaphor by allowing users to define the format of *control panels* interactively, so that a control panel can itself be designed using a control panel.

The user interface work in the Aspect project (Ref. 7) has been more fundamental and has involved the development of an autonomous display manager. This allows for the rapid construction of user interfaces and has, in fact, been demonstrated by developing a direct manipulation system for the Aspect data model.

Smalltalk-80 (Ref. 8) is an environment which by our criteria must be considered a programming environment rather than an IPSE, but it shares two important features with ISM. These are its object-oriented architecture and the intimate relationship between the environment and its user interface. Smalltalk's designers realised that if the full power of the environment and its language were to be properly exploited then a powerful and fully consistent user interface would be required.

In Smalltalk-80, interaction with the environment is performed through the browser. The browser allows objects within the system to be viewed both in terms of their internal representations and their external interfaces. Writing programs in Smalltalk-80 is achieved by using the browser to define new classes of objects as sub-classes of existing classes. Smalltalk-80 has proved to be very effective for the rapid prototyping style of programming, largely as a result of the powerful and consistent user interface.

These user interfaces are examples of the direct manipulation style of interaction (Ref. 9). Direct manipulation interfaces allow items to be displayed continuously for the duration of the user's interest in the items. Instead of having to know a complex command syntax, users perform physical actions like pressing a button, or selecting a menu item with a pointing device. When some operation has been carried out on an object, the effect is immediately visible to the user.

For a large class of interaction operations, direct manipulation reduces the amount of syntactic knowledge which users must possess, permitting them to concentrate on the task domain semantics. Users of well designed direct manipulation interfaces are given the impression of being inside a model world (Ref. 10) represented by the interface itself. By contrast, the command language interaction style employed by many environments requires that users know not only what they wish to do, but also the syntax of the commands required to make it happen. The command language is a medium through which users and the system have a conversation about abstract entities not explicitly represented on the user's screen.

The ISM user interface uses a *dynamic forms* metaphor to permit direct manipulation of ISM objects. Objects are viewed as forms, where an object's attributes are represented as form fields and permissible actions which can be performed on the object are displayed as buttons. Objects may embody relationships which cause automatic propagation of values across attributes, and it is this feature that provides the dynamic properties of forms within ISM. New objects and object classes are themselves created by filling in forms. In addition to imposing a highly consistent style of interaction, dynamic forms provide a useful framework for encapsulating domain-specific knowledge which can be used to provide an appropriate degree of automation.

3 An object-oriented IPSE architecture

The ISM environment has an actor-based architecture which maps naturally on to its functional view of a collection of independent, co-operating agents. Agents are asynchronous processes holding responsibility for different aspects of poject support, and include ISM users who are treated as human agents. Agents respond to messages from other agents, by invoking behaviours to perform some action. Messages may arrive for processing in any order, and a behaviour responding to a message may involve issuing messages to other agents. The response time of an agent's behaviour may vary greatly, and in the case of human agents be of the order of months for certain tasks.

ISM agents embody knowledge local to their domain of responsibility. Changes to the state of the environment are performed by the application of this knowledge to the symbolic manipulation of project artefacts. An object-oriented programming paradigm has been adopted for the implementation of project artefacts. These ISM objects encapsulate both local state data (*attributes*) and the operations which may be performed on that data (*methods*). Hierarchical relationships between classes of objects are represented by inheritance networks; attributes and methods defined for an object class are inherited by any sub-classes of that class.

An object-oriented extension to Prolog (ISML) is used as the ISM implementation language. Among Prolog's distinctive features are its declarative style, which permits knowledge bases to be rapidly prototyped using facts and rules. This characteristic is exploited by those components of ISM agents which perform logical inferences to reason about project artefacts and determine appropriate reactions to changes of the project state. ISML's object-oriented extension imposes a partitioning of the otherwise flat Prolog name space and enforces a discipline of manipulating bodies of related data as objects instead of as arbitrary sets of facts and rules.

ISM is implemented on top of the UNIX operating system running on a network of Sun-3 workstations. ISML is used as an inference engine and a knowledge base, but there are classes of activity for which its use is inappropriate. Many existing Sun and UNIX tools are integrated into the environment both as individual agents and as components of agents. Examples of these include C compilers, mail transfer programs, window management systems etc.

The aim of the ISM project was to demonstrate the feasibility of constructing an environment based on 'intelligent' agents. Practical considerations (performance, space utilisation etc.) were of secondary importance. In general, the implementation of an object store in an applicative language like Prolog is probably too inefficient for large-scale use. However, the direct manipulation system described here could be readily supported on top of any object store.

3.1 The ISM federation

The ISM federation consists of a collection of active agents, each responsible for some aspect of project support and for managing a part of the ISM knowledge base. These include a project management agent, a configuration management agent, a communication agent etc.

Project artefacts are themselves ISM objects which are logically passed around inside the ISM system. When a user writes a piece of C code for example, an instance of the class *c_source_code* is created. To announce a project meeting, an instance of the class *meeting_announcement* is created, and this has very different processing requirements than the *c_source_code* object.

It is intended that ISM will possess enough knowledge to apply the appropriate agent(s) to an object and that agents will embody sufficient local knowledge to apply themselves in the correct way. Thus a compiler agent would invoke the appropriate compiler depending on the language of the code to be compiled. Should the attempt to compile fail, then an object containing the information relating to the failure would be created and passed to the user who submitted the object containing the source code.

ISM users are treated as members of the set of agents, so that to the ISM knowledge base they will look like any other agent. Consider a project member Fred, who might be a member of the class *user* with the following attributes:

Name	:Fred Smith
Staff number	:lu27
Grade	:programmer
Languages	:ada, c, pascal, prolog
Projects	:ISM2, nimrod aew
Holiday	:03/06/88–19/06/88

A *mailer* agent incorporates knowledge about mail distribution. Given user information, the ISM mailer would infer that a meeting announcement for (say) ISM2 which does not fall within the period 3rd to 19th of June 1988 should be sent to Fred and any other user with appropriate attribute values. Whoever issued the meeting announcement can be relieved of the task of explicitly sending copies to all interested parties and of storing a file copy.

4 Manipulating objects using dynamic forms

The principal characteristic of the ISM user interface is that it integrates users with the environment. Whether an ISM agent is actually a person or a program is largely transparent to the environment because the same communication protocols are used for both.

Users interact with the environment by directly manipulating objects within it. This means that a user does not have to remember a host of different command styles for different operations. In the UNIX C shell, for instance, users are expected to know which arguments, if any, are associated with particular commands. Experiments undertaken as part of the ISM work† suggest that complex command languages such as those used by UNIX are deficient in this respect, especially among novice users, or those who are unable to invest a great deal of time in familiarising themselves with a particular interaction style.

The ISM user interface approach has been to use a declarative rather than a command-driven style of interaction. To initiate some action, a user supplies information to the system and ISM decides what to do with it. To avoid natural language processing, some structure has to be imposed on the user-supplied information. Dynamic forms structure the way users interact with ISM while providing a framework which can exploit built-in knowledge.

Form-based user interfaces are not a new idea (Ref. 11), and have been used in data processing systems for many years. Recent research has been devoted to using them as front ends to systems which run on machines equipped with high-definition bit-mapped displays and pointing devices. Cousin-Spice (Ref. 12) is an example of such a system; Hayes describes how information in a form is structured into two types of field. These embody information representing commands, and that representing parameters. The former type of field (henceforth referred to as a button) merely needs to be *selected* by the pointing device to execute the command (*pressing* the button), while the latter requires a *value* to be associated with it by typing into a space adjacent to the field's label.

Ordinary form systems do not, however, exploit the structure imposed on information to the degree that dynamic forms can, although simply providing users with a set of fields to indicate the extent of information required can often be helpful.

Dynamic forms are similar to what, in the Information Lens system (Ref. 13), are called semi-structured messages. Malone observes that these semi-structured messages 'enable computers to process a much wider range of information than would otherwise be possible'. The structure imposed by the forms, as a framework, can encapsulate much information in fields which would otherwise have to be extracted by parsing of free text. It is relatively simple, given the structure of a form *type*, to express interdependencies between fields as if the form were a spreadsheet. This feature enables much automatic processing of information.

Dynamic forms' power of direct manipulation stems from the fact that they are merely the physical representation of ISM objects. Thus a user filling in a form is actually creating an object. Any object visible to the ISM knowledge base can be viewed as a dynamic form.

As suggested above, forms are typed; i.e. a different form exists for each kind of operation. One would expect that a form used to build an object module from the most recent versions of its components would have a different set of fields and buttons from a form used to announce a project meeting. Form types can be, and usually are, defined hierarchically. For instance, the meeting_announcement form type is a specialisation of the more general mail type. This ability to define form types incrementally maps neatly on to the class inheritance features of ISML in which forms are implemented.

New form types can be defined by any ISM user, simply by defining a new object class. Form types within existing data processing systems are typically defined by a system administrator, and ordinary users are restricted to using what they are provided with. Within ISM, however, it is recognised that individual users may not only have very specific personal requirements, but may also make a valuable contribution to providing a rich set of form types for particular applications across the ISM system.

The meeting_announcement form type, introduced above, is defined as an object class, and is analogous to an empty form template. Fig. 3 illustrates the creation of an instance of such a form. By filling in a meeting announcement form we effectively create an instance of that class. By filling in a field value we assign values to the attributes, and by pressing a button we send a message which invokes the corresponding method.

The ISM dynamic forms interface can be summarised as follows:

- Dynamic forms are typed, abstract entities representing objects within ISM. There is a one-to-one mapping of forms to objects, where a form is the physical representation of an object on a user's workstation screen. To interact with a form is to directly manipulate the form's underlying object. New form types can be defined hierarchically, with new types inheriting features from their super-types.
- Fields are typed, and can be assigned default values. Inter- and intra-object relationships between fields (attributes) can be expressed. This permits ISM to attempt to complete as much of the form as possible, as and when the information becomes available, and to oversee a user's interaction with an object in real time.
- Form buttons map on to object methods. Methods represent an object's external procedural interface. A method is essentially a procedure which may be called by sending the owning object a message; thus pressing a button on a form at the user interface results in a message being sent to the underlying object in the ISM object store. Syntactically, a message consists of a selector, corresponding to the name of the method to be called, followed by a list of one or more

† The experiment took the form of a questionnaire which asked UNIX users questions about how they used their systems. The questionnaires were given to a broad spectrum of users, from novices to 'gurus' both within academia (Keele and Lancaster Universities) and industry (Software Sciences Ltd.). The subject evidently arouses strong passions among all categories of users, but most expressed some dissatisfaction with the interaction style.

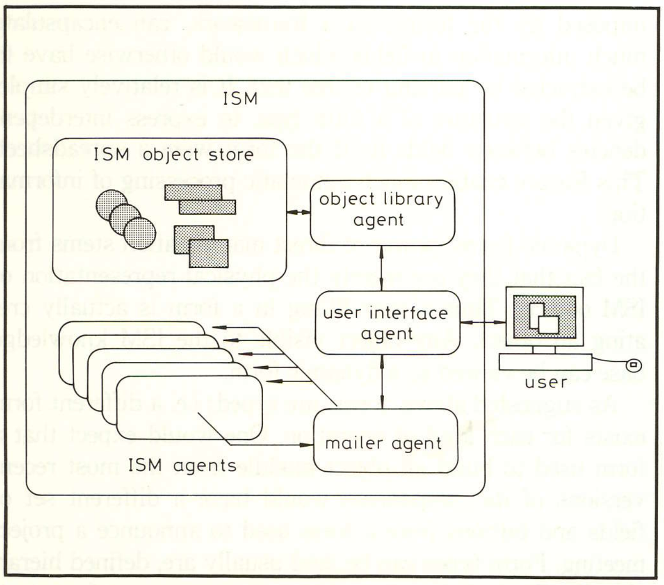


Fig. 1 Logical view of the user interface's relationship to ISM

arguments. Arguments are typed and declared with the object method definition. Argument values are context dependent and their type declarations serve to indicate the range of values which may be assigned. The argument typing scheme is necessary to assist the resolution of argument values at run time. Users are, as far as possible, protected from the need to know about the details of a method declaration in terms of arity and argument types in order to call the method. Examples of method argument types include:

- ☐ the id of the object to which the message is sent
- ☐ the value of one of the object's attributes
- ☐ a value which is not embodied by the object's name or attributes (for example, an object of the class *document* may have a *print* method with an integer argument specifying the number of copies required; a textual string giving an English description of the criteria constraining the range of argument values is included in the argument type declaration; this is used to prompt the user for an argument value via a pop-up form in the event of the method button being pressed).

4.1 ISM user interface functionality

The principal requirements of the ISM user interface are that users must be able to browse the object store and manipulate its contents. In other words, the user needs to be able to find out what is in the object store, look at individual objects, create new objects, modify existing objects (but only if permitted to do so) and define new object classes.

Three components implement the ISM user interface:

- The user interface agent itself is responsible for physically presenting objects on the screen, handling user input, mouse clicks etc. It performs the translation of objects into forms based on the simple attribute-to-field, method-to-button mapping.
- The object library agent is essentially a server to the user interface. Requests from the user for information about some object or group of objects is relayed by the user interface agent to the object library. The object library searches the object store, collects information about the required object(s) and returns information regarding class instances, object attributes and methods etc. to the user interface. By constraining the way operations are performed on objects by the user interface, the object library ensures that the integrity of the object store is maintained and that the user interface is protected from unpredictable behaviour such as Prolog backtracking.
- The mailer agent is responsible for what happens to objects once they have been created or modified by the user interface. As an example, an instance of the class *compile* would be sent to the compiler agent for processing. This is a logical view of course; in fact the mailer would send a *compile* message to the compiler agent with parameters unified with attribute values of the instance of the compile class — source code location, compiler options etc.

Fig. 1 represents a logical view of the relationship of the ISM user interface to the ISM as whole.

Browsing the object store:

Browsing the object store can be done on two levels: a definition level and an instance level. The definition level provides information about what classes exist, whereas the instance level permits the manipulation and creation of particular class instances. To date, work has been concentrated at the instance level.

Browsing at the instance level enables users to see what instances of a class exist. A user can view individual instances as completed forms, and create a new instance by filling in a blank form.

Fig. 2 illustrates an example of browsing at the instance level. The narrow window labelled **ISM user interface**, containing the three buttons labelled **Browse**, **Define** and **Quit**, represents the top-level ISM control panel:

- The **Browse** button permits objects within the object store to be manipulated, and objects representing new instances of existing object class definitions to be created.
- The **Define** button permits the definition of new object classes.
- The **Quit** button terminates the user's dialogue with the system.

In the scenario represented by Fig. 2, the user has pressed the **Browse** button, causing the user interface to send a message to the object library requesting an instance of the class *view_instances* to be created and displayed as a form. The object class view_instances is designed specifically to facilitate the viewing of instances of other object classes. Once created by the object library, the instance of view_instances resides within the ISM object store until the user interface instructs the object library to delete it.

The instance of view_instances is represented by the form labelled **view_instances: view_instances_0** which partially overlays the top-level control panel. Because objects of this class exist only for the duration of their display, users are not required to assign them names. Instead, ISM generates unique identifiers of the form *view_instances_*$\langle n \rangle$.

The form **view_instances: view_instances_0** consists of two sub-windows:

- A control panel containing buttons representing the messages defined for the class view_instances. These are the methods **close**, **delete** and **send**, which are generic to all ISM objects and have the following functionality:

- ☐ With the **close** method, object classes such as view_instances, which are designed for the purpose of communicating information between users and the environment, are defined to be a subclass of *display_class* in which the definition of the **close** method is unified with that of the **delete** method. This is not the case with ISM object classes representing more persistent items of data in which the method merely removes the form from the display.
- ☐ The **delete** method removes the form representing the object from the display and deletes it from the ISM object store, releasing the space which it occupied. The operation of this method is subject to constraints which determine whether deletion of the object is permissible.
- ☐ The **send** method removes the form representing the object from the display and dispatches it to the mailer for appropriate processing by other ISM agents.

The methods **create_instance** and **view_instance** are described in the sections on creating and viewing an instance.

- The lower sub-window contains four fields: **instances**, **object_class**, **subclasses** and **superclasses**, corresponding to attributes defined for the class view_instances. The icons labelled **set** (a train set) and **unique** (a large number one) adjacent to the attributes indicate their type. Attributes of *unique* type may have a maximum of one value; *set* types may have zero or more non-duplicated values. Two additional classes of attributes exist: **bag** and **bigtext**. An attribute of type *bag* may have zero or more, possibly duplicated, values. *bigtext* attributes are used for unstructured attributes — source code, mail text etc. By default, the first value of set or bag type attributes is displayed by the form. Users may traverse the list of values by clicking a mouse button on the attribute name. Alternatively, a full listing of the values may be viewed by selecting the **view all** option from an attribute's menu. Similar facilities exist to permit the viewing of bigtext attributes.

Viewing an instance:

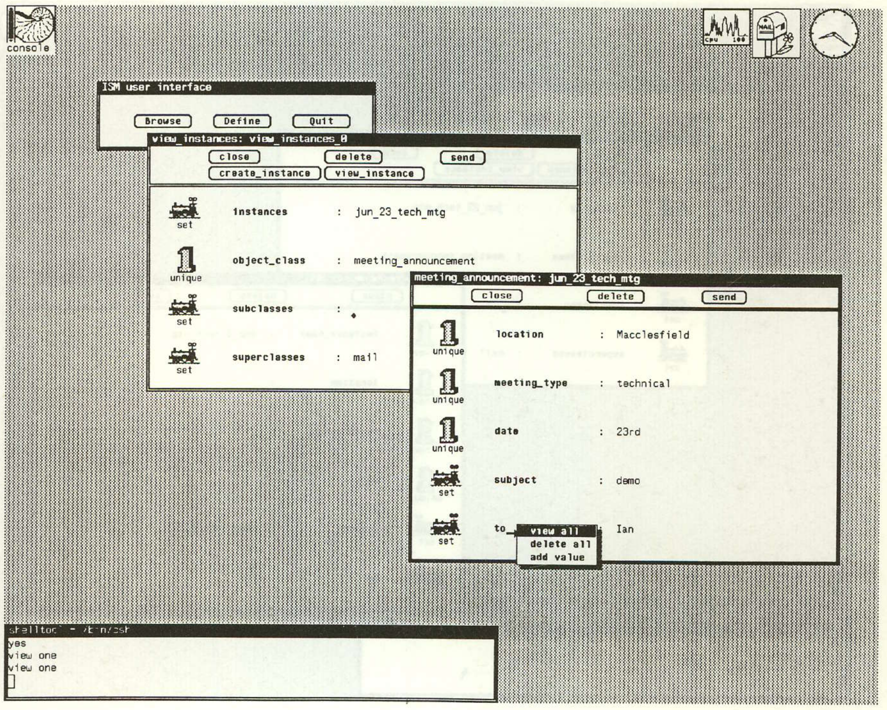
The example illustrates a stage of a user's interaction where values have been assigned to attributes, and the method view_instance has been invoked.

When first created, no values are associated with an instance of the class view_instances, and the user is presented with a blank form. In the example, the user has assigned a value to the object class attribute, indicating that he wishes to inspect instances of the class *meeting_announcement*.

Constraints associated with object_class embody relationships between the other three attributes and enable values of the other three attributes to be automatically inferred. On receipt of the value of object_class, the following values are generated:

- The attribute superclasses has been assigned the value *mail*, which represents the sole superclass of the meeting_announcement class.
- The *null* value has been generated for the subclasses attribute, indicating that meeting_announcement represents a leaf of the inheritance network.
- A set of values for the instances attribute have been generated. These represent instances of the class meeting_announcement resident within the ISM object store. The value *jun_23_tech_mtg* is currently visible.

In the example, the user has opted to inspect the jun_23_tech_mtg object by clicking on the view_instance button. The message *view_instance* has been sent to the object library. The object library has resolved the value of the method's single argument to be the currently selected value of the instances attribute and sent the message



set	instances	: jun_23_tech_mtg
unique	object_class	: meeting_announcement
set	subclasses	: •
set	superclasses	: mail

unique	location	: Macclesfield
unique	meeting_type	: technical
unique	date	: 23rd
set	subject	: demo
set	to	: Ian

```
yes
view one
view one
```

Fig. 2 Viewing an instance of the class meeting_announcement

'view_instance(jun_23_tech_mtg)' to the view_instances_0 object residing within the object store. view_instances_0 has responded by searching the object store for the required object and returning its details to the user interface agent via the object library.

On receipt of the message containing the object's details, the user interface agent has mapped them onto the form labelled **meeting_announcement: jun_23_tech_mtg** in the bottom right-hand part of Fig. 2.

The **to** attribute belonging to the jun_23_tech_mtg form illustrates the menu options associated with set and bag attributes. The options available are **view all**, **delete all** and **add value**. These supplement the editing operations which may be directly applied to a selected value — namely to delete or modify it.

Whether such editing operations on values may be accepted by the object library is dependent on the constraints associated with the attribute definitions. Some objects (for instance those representing archive components of a project's milestones) may be immutable. An attempt to modify such an object would spawn a warning message form.

Creating an instance:

Fig. 3 illustrates the creation of a new instance of the class meeting_announcement by invoking the create_instance method. This causes a blank form labelled **meeting_announcement: NULL** representing the class meeting_announcement to be displayed. The example shows the form in a state of partial completition by the user.

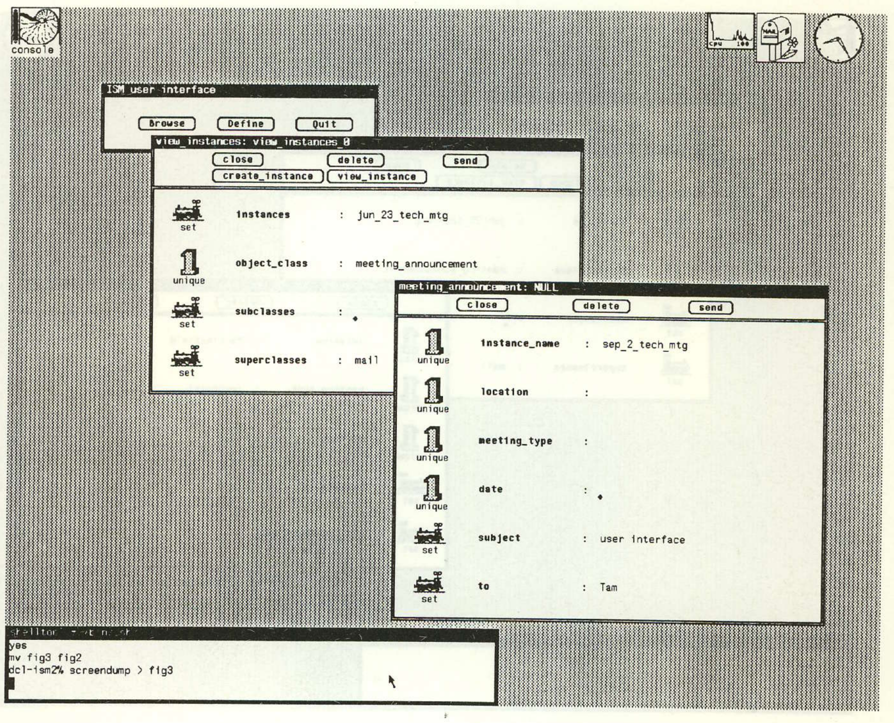
Note that an additional field, **instance_name**, is displayed in the attributes sub-window of the meeting_announcement class form. Objects of the class meeting_announcement are persistent in the sense that they exist for longer than the duration of the user's dialogue with the environment. The user is therefore required to assign the object an identifier before invoking the **send** or **quit** methods.

The **subject** attribute again illustrates dynamic forms' ability to express inter-attribute constraints which automatically infer values of attributes from user-supplied attribute values.

Constraints associated with the subject attribute have used the new subject value to parametrise a consultation of the ISM knowledge base and so infer the set of project members to which the message should be sent. These have been duly assigned the **to** attribute. It is recognised that such inferred values may not be correct in every context, and (as with any attribute value) the user has the power to edit them, provided that the values are not explicitly constrained to be immutable. In the case of the **to** attribute, values may be added or deleted, but the system will complain if an unknown name is added to the set.

On completion of the form, and assuming (as will usually be the case) that the user wishes the new object to be processed immediately, the send button is pressed. The object library creates an object with identifier and attribute values as allocated by the user. The object is then sent to the mailer agent, which holds responsibility for forwarding it to the appropriate agent(s) for processing. In the case illustrated, this would involve sending the object's details to each of the project members held by the **to** attribute and a copy to a *project_history* agent.

Users are not constrained to assign values to every attribute when creating an object, but will be requested for further information if that supplied is inadequate. It is left to the various processing agents to attempt to make the best sense of information encapsulated by an object's attributes. This is an issue being addressed by the design of the



view_instances: view_instances_0

```
instances      :  jun_23_tech_mtg
object_class   :  meeting_announcement
subclasses     :  ◆
superclasses   :  mail
```

meeting_announcement: NULL

```
instance_name   :  sep_2_tech_mtg
location        :
meeting_type    :
date            :  ◆
subject         :  user interface
to              :  Tam
```

```
yes
mv fig3 fig2
dcl-ism2% screendump > fig3
```

Fig. 3 Creating an instance of the class meeting_announcement

ISM mailer agent (Ref. 14). The mailer agent provides a framework capable of encapsulating the knowledge required to reach decisions about the appropriate processing of different objects.

Defining a new object class:

The ISM user interface exploits the object-oriented class inheritance mechanism for the incremental definition of new object classes by permitting the sharing of code across classes. The possession of the generic methods, **close**, **delete** and **send** by all ISM objects is an example of this form of re-use.

Fig. 4 shows an example of a form representing an instance of the class *class_definer* used to create new class definitions. The example illustrates the definition of a new object class initiated by pressing the **Define** button on the top-level ISM control panel. In the scenario, the class *tech_meeting_announcement* is being defined as a specialised case of the more general meeting_announcement class.

The object class_definer_0 has eight sub-windows, described in turn from the top:

- The two top-most sub-windows represent the usual control panel of method buttons and a sub-window containing those attributes specific to class_definer instances.

The **close** and **delete** methods have the usual effect. The **send** method will cause the information held by the form to be used to (attempt to) define a new object class. If successful, the class_definer object will be deleted from the object store; otherwise, it will be retained for debugging purposes.

The **remove** method will selectively delete any of the new class' method or attributes.

Attributes **class, superclasses** and **subclasses** are all relative to the new class being defined. A user must assign a value to class, but the other two may be left with no value. If one or more values are assigned to superclasses, they must all be the names of existing classes. In this case the new class will inherit all the superclasses' attribute and method definitions unless explicitly overloaded.

- The next two sub-windows on the right-hand side of the form are reserved for the methods and attributes which will be defined for the new class.

The upper left-hand sub-window labelled **methods** is used to create method definitions for the new class. By clicking on the button icon, a button will appear in the adjacent sub-window. The user may then type the method's selector (name) on the new button. In the example, a method **book_room** has been defined.

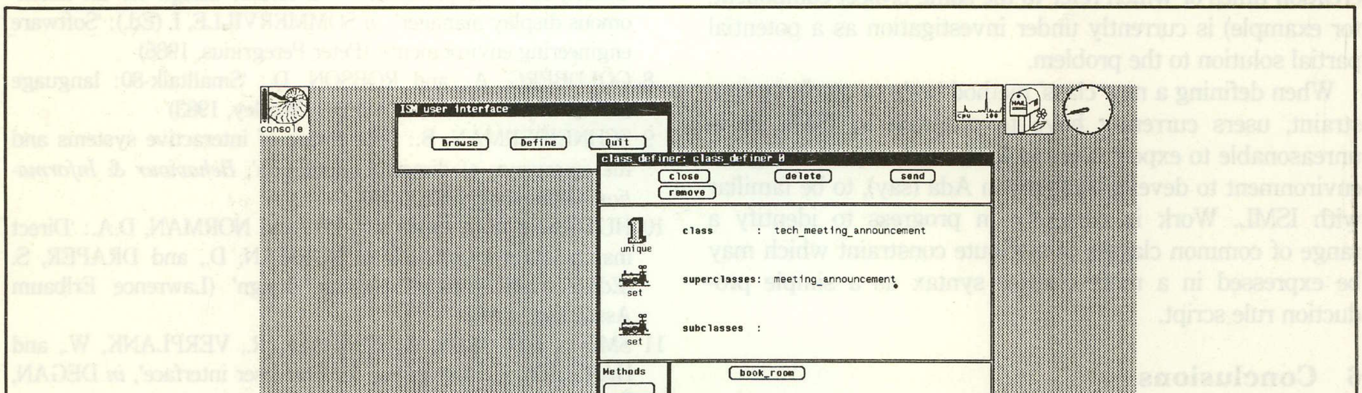
The left-hand sub-window labelled **atributes** contains the four attribute type icons. The user can define an attribute by clicking on the icon of the desired type.

In the example four attributes have been defined: **subject**, **description**, **location** and **time**. Note that attributes subject and location are already defined for the class meeting_announcement, but these are overloaded by the new definitions for all instances of the new class.

- The bottom left and right sub-windows are provided to permit the user to define method bodies and attribute constraints.

In the example, the user has selected the subject attribute with the mouse. The user interface has interpreted this as a desire to define a constraint for the attribute and emboldened the attribute name. The smaller of the bottom sub-windows indicates the current status of its larger neighbour; in this case that a constraint for the subject attribute is being defined.

The larger sub-window has enhanced text editing facilities. The example illustrates a portion of ISML code defining an attribute constraint designed to automatically infer



```
Subject::infer_location(Subject).

infer_location(Subject):-
    done_at(project_model,Subject,Site),
    atom(Site),
    infer_value(location,Site).
infer_location(_)
```

```
shelltool - /bin/csh
[2] 7964
[1]    Done                    ISM
dcl-ism2% screendump > fig4
```

Fig. 4 Defining a new object class

the value of the location attribute on receipt of a value for the subject attribute.

5 Future development

At present, browsing the object store at the definition level merely allows the user to view a list of all extant object classes. As the number of object classes belonging to a project grows, users may experience difficulty identifying those which are of interest. For example, a user may suspect that a class exists which embodies most of the functionality he requires but may not know where to look for it or what it is called.

The user interface should provide a mechanism to permit users to describe the features they require and perform some pattern matching in an attempt to identify a sub-set of potentially useful classes. As an example a user may issue a request of the form:

'show me an object class possessing attributes [name, grade] and methods [assign]'

which would return a list of classes with one or more of the attributes/methods specified.

This mechanism would additionally provide enhanced assistance for browsing at the instance level:

'show me an object possessing attributes [[name, Fred], [grade, programmer]]'

However, in the absence of any method of focusing the user interface's search of the object store, the mechanism would incur prohibitive performance overheads. The ability to define hypertext-like links between heterogeneous but related objects (objects of different classes but with similar creation times or which refer to the same project component, for example) is currently under investigation as a potential partial solution to the problem.

When defining a new class' method body or attribute constraint, users currently have to program in ISML. It is unreasonable to expect users of ISM, who may be using the environment to develop systems in Ada (say), to be familiar with ISML. Work is currently in progress to identify a range of common classes of attribute constraint which may be expressed in a more concise syntax as a simple production rule script.

6 Conclusions

This paper describes a user interface to an IPSE based on an object-oriented programming paradigm. The power of the user interface is derived from the simple mapping of forms onto objects.

By treating every object as a form, the user interface provides a simple and consistent interaction metaphor which embodies a degree of built-in automation and avoids parsing of free text. Users directly manipulate the contents of the environment instead of conversing with it through an abstract command language.

The architecture of the environment is one of a federation of co-operating agents. A high degree of integration is achieved by modelling a project as a process and by the adoption of a uniform communication paradigm between all the environment's agents. Users are considered to be members of the environment's federation of agents, each of which have an explicit set of responsibilities.

Project artefacts are embodied as objects and include messages between users, from the environment to users, and from users to the environment. Objects are logically passed around inside the environment for processing by appropriate agents. New objects are created simply by the process of filling in a form in which the environment performs dynamic constraint checks and attribute value inference.

7 Acknowledgments

Thanks are due to our collaborators in the ISM project, Software Sciences Limited and the University of Keele. The research work is funded by the Alvey Directorate, UK.

8 References

1 WINOGRAD, T.: 'Breaking the complexity barrier (again)'. Proceedings of ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages — Information Retrieval, Gaithersburg, MD, USA, 1973

2 BOURNE, S.R.: 'The UNIX system' (Addison-Wesley, 1982)

3 HALL, J.A., HITCHCOCK, P., and TOOK, R.: 'An overview of the ASPECT architecture', *in* McDERMID, J. (Ed.): 'Integrated project support environments' (Peter Peregrinus, 1985)

4 ALDERSON, A., BOTT, M.F., and FALLA, M.E.: 'An overview of the ECLIPSE project', *in* McDERMID, J. (Ed.): 'Integrated project support environments' (Peter Peregrinus, 1985)

5 REID, P., and WELLAND, R.C.: 'Project development in view', *in* SOMMERVILLE, I. (Ed.): 'Software engineering environments' (Peter Peregrinus, 1986)

6 ENGLAND, D.: 'A user interface design tool'. Proceedings of First European Software Engineering Conference, Strasbourg, France, 1987

7 TOOK, R.: 'The presenter — a formal design for an autonomous display manager', *in* SOMMERVILLE, I. (Ed.): 'Software engineering environments' (Peter Peregrinus, 1986)

8 GOLDBERG, A., and ROBSON, D.: 'Smalltalk-80: language and its implementation' (Addison-Wesley, 1983)

9 SCHNEIDERMAN, B.: 'The future of interactive systems and the emergence of direct manipulation', *Behaviour & Information Technology*, 1982, **1**, (3)

10 HUTCHINS, E.L., HOLLAN, J.D., and NORMAN, D.A.: 'Direct manipulation interfaces', *in* NORMAN, D., and DRAPER, S. (Eds.): 'User centered system design' (Lawrence Erlbaum Associates, 1986)

11 SMITH, C.D., IRBY, C., KIMBALL, R., VERPLANK, W., and HARSLEM, E.: 'Designing the Star user interface', *in* DEGAN, P., and SANDEWALL, E. (Eds.): 'Integrated interactive computing systems' (North-Holland, 1983)

12 HAYES, P.J., and SZEKELY, P.A.: 'Graceful interaction through the COUSIN command interface', *International Journal of Man-Machine Studies*, 1983, **19**,

13 MALONE, T.W., GRANT, K.R., and TURBAK, F.A.: 'The Information Lens: an intelligent system for information sharing in organisations', *in* OULSON, M. H. (Ed.): 'Technological support for work group collaboration' (Erlbaum, 1988) (in press)

14 RODDEN, T., and SOMMERVILLE, I.: 'Mailtrays: an object oriented approach to message handling'. Report, Department of Computing, University of Lancaster, Lancaster, England, 1987

P. Sawyer and Prof. I. Sommerville are with the Department of Computing, University of Lancaster, Bailrigg, Lancaster LA1 4YR, England.