Interacting with an active, integrated environment

Thomas Rodden, Pete Sawyer, & Ian Sommerville
Department of Computing
University of Lancaster
Lancaster, LA1 4YR, UK.

Abstract

Software engineering environments are intended to provide a cohesive and integrated set of tools to support the process of software engineering with much current research into environment design focussed on maximising the degree to which these tools can be integrated. This paper describes the architecture of a prototype environment which attempts to achieve a high degree of integration using techniques drawn from artificial intelligence, office automation and object-oriented programming. This environment is implemented as a federation of intelligent, co-operating agents which communicate, with each other and with users, by message passing. This paper is particularly concerned with user interface integration including the mechanisms employed to permit inter-agent and agent-user communications.

1. Introduction

In common with any engineering task, effective software engineering requires the use of appropriate tools. In recognition of this fact, much effort has been invested in producing software tools such as compilers, symbolic debuggers, program analysers and so on. The concept of programming environments arose from the idea of collecting such tools together into kits where users could apply them at appropriate stages during the course of software development.

UNIX [Ritchie 1978] is perhaps the best-known example of such a programming environment. The Unix system provides a variety of different tools but, more importantly, provides tool integration mechanisms (character files, I/O redirection, pipes, shell programming) which allows tools to be used in concert. Outputs from one tool can serve as inputs to others. Thus, powerful tools and tool sequences can be built by putting relatively simple tools together.

In programming environments like UNIX, the software developer must take full responsibility for the application of the correct tools to the components of the development process. The environment itself encapsulates little information regarding interdependence of tools and components beyond that which can be expressed in Makefiles and shell scripts. Furthermore, the basic Unix toolset is principally intended for programming support and it does not provide a great deal of support for other software process activities.

A project support environment differs from a programming environment by providing support across a broad spectrum of project activities, 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 forming part of a project *process*, can it be said to be an *integrated project support environment* . This term and its acronym (IPSE) has been widely adopted in Europe but has gained less currency elsewhere. An equivalent term, perhaps more widely accepted in North America, is software engineering environment (SEE) and the terms IPSE and SEE may be used interchangably.

ISTAR [Dowson 1987], ECLIPSE [Alderson 1985], ENCORE[Zdonik 1985] and SAGA/ENCOMPASS [Campbell 1986] are examples of software engineering environments which address the problem of supporting the whole software development process with an integrated set of software tools. In these systems, 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 degree which was not possible in older, file-store based environments. A user interface based on bit-mapped workstations and direct manipulation is supported.

Current generation environments are "passive" IPSEs which exhibit two levels of integration:

- Data integration via a database management system
- User interface integration via standards and tool support

The systems are passive because they support rather than participate in the development process. They do not support activity integration inasmuch as the activities involved in the software process are initiated entirely by user actions. In an active system, as well as data and interface integration, activity integration is also supported. Information regarding interdependences of tools and components may be used to initiate actions automatically when some conditions are met. These condition may be triggered by external events such as the delivery of some document, by time or by specific internal states.

Recommended by: *Leon Osterweil*

To be most effective, an active, integrated environment must be driven by a model of the software process and current work is focussed on understanding this process and establishing notations to define process activities and their coordination [Curtis 1987]. The work in this area is not yet sufficiently mature for adoption in a system such as that described here so the driver for our prototype system is not a complete process model but rather a management-oriented view of a software project. This is captured in a system knowledge base. However, the structure of the knowledge base and the system architecture is sufficiently flexible that, as our understanding of the software process improves, more and more process modelling information may be incorporated and used to drive development activities.

This model encapsulates the knowledge which, early in the history of programming environments [Winograd 1973], was identified as being necessary for tackling the problem of complexity in large software engineering projects. Information about a project's, aims, products, resources, timescales, etc. can be captured in the knowledge base. This information is embodied within the IPSE as objects. By manipulating these knowledge sources the environment is able to reason about an evolving project, integrate the various transformations which need to be applied to its components, and thus automate some software process tasks. The ability to reason about a project, to initiate actions and generate transformations automatically embodies the IPSE with the attribute of being *active* .

We believe that a key characteristic of an active, integrated environment is the user interface which, as well as being integrated and consistent in its own right, must be tightly integrated with the environment. Indeed, this is one of the great strengths of programming environments such as Smalltalk[Goldberg 1983] and Cedar[Teitelman 1985] and a definite weakness of 'open' environments such as ISTAR [Dowson 1987] where a tightly integrated interface is difficult to achieve. We believe that the approach we have adopted goes some way towards bridging the gap between these approaches and much of this paper is devoted to a discussion of how we have tackled the problem of user interface integration.

2. An active integrated environment

In most current environments, tools are layered around some database system and users interact directly with those tools. Efforts have been made in both the USA and in Europe to develop database kernel standards (PCTE and CAIS) and these standards are likely to form the basis of environment products in the next few years. However, the kernel approach has the disadvantage that the database structure limits the overall functionality of the environment ; it is particularly difficult to integrate AI-based approaches with this structure. We believe that the kernel model of an environment is a constraining one and that progress towards more intelligent environments requires this model to be discarded.

By contrast, our model of an active IPSE, which is called the ISM, has neither tools nor a kernel in the accepted sense. The ISM consists of a federation of co-operating, distributed agents (Figure 1) which embody sufficient contextual knowledge to be invoked on an opportunistic basis. Agents are autonomous, asynchronous, immutable and may be cloned on several nodes in a network.

An agent encapsulates both local data and the operations which may be performed by the agent and presents an external interface consisting of the set of allowed operations. Operations on data held by an agent are initiated by sending a message to the agent requesting some action. Controlled access of data allows sharing of information required by an diversity of agents.

While the set of agents contains elements which may be thought of as *smart tools*, an intelligent configuration management agent for example, it also includes users of the environment. Current AI technology permits human expertise to be mimicked in only very limited domains, and hence creative input to any software engineering project must still come from humans.

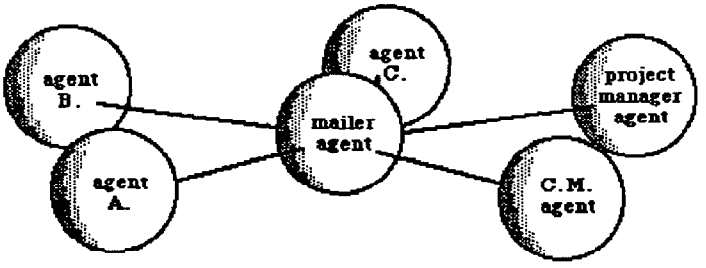

Figure 1. Logical view of active agent topology.

The integral part played by human agents does not compromise the environment's claim to being active. If all agents, both software and human, are properly integrated, then they will all appear to the environment to have a consistent external interface. This enables the automatic scheduling of all agents to perform specific, project-related tasks, as black-box processes. As technology develops, it may turn out that the only significant distinguishing characteristic of human agents is a somewhat greater response time than that of many of the software agents.

A powerful attribute of the ISM, which is enabled by its integrated architecture, is the ability to delegate. Unlike most current environments whose tools operate largely in isolation, many complex tasks can be performed automatically by agents co-operating. Thus a complex operation may be disassembled into subtasks, each of which is delegated to the agent with the appropriate ability to perform the subtask.

For example, the design of an important program module may be running late. A consequence of this could be that the slippage will have to be absorbed by temporarily reallocating other members of the design team to work on the module. Finding an acceptable solution to this problem which

minimises the resulting knock-on effects will involve the co-operation of the project management agent, a planning agent, a scheduling agent, and one or more human agents.

The above example illustrates the requirement that agents must be asynchronous processes capable of running concurrently. The project management agent would not be able to cause the rescheduling of the project until the planning agent had established a preferred reallocation of resources (the execution time of a rescheduling process may be of the order of days if human agents are involved), yet it may be required to respond to other events during the intervening period.

In principle, any agent may call upon any other agent to perform a task by sending it a message. This implies that agents must know about the services offered by other agents so that messages are not sent to agents which do not have appropriate operations. The approach which has been taken is to devolve this knowledge to an intelligent message handling agent (the mailer) which embodies knowledge about the functionality, location, availability and interface of other agents.

An agent requiring a service from some other agent sends a message via the mailer agent which routes the message to an agent which can respond to the request. In addition, the mailer agent allows users (human agents) to initiate actions within the IPSE in the same manner as any other agent. It also allows other agents to request services of a user as they would of any agent.

In addition to being able to initiate actions within the ISM by sending messages to agents via the mailer, a user interface (the object browser) based on direct manipulation [Schneiderman 1982] permits users to access the contents of the knowledge base. The object browser controls knowledge-base access and co-operates with the mailer to translate user requests into the message format appropriate to the agents responsible for the individual items of data being accessed. The principle of protecting data from unauthorised access is therefore enforced consistently for all agents, human and otherwise.

Knowledge is held within the partitioned system knowledge base as objects. Objects [Rentsch 1980, Wegner 1987] were selected as an appropriate knowledge representation mechanism for the ISM using the same rationale employed in the selection of agents to embody the active elements of the environment. Objects enforce the principle of information hiding, encapsulation of data and procedures, and provide a conceptually elegant means of packaging information.

Objects within the IPSE knowledge base encapsulate both attributes, data private to the object, and methods, the operations which may be applied to the data. An object is a named instance of a class where a class can be thought of as a template definition of an object type. Objects communicate by message passing, where messages invoke object methods. Multiple inheritance is supported.

Attributes may have class default values, are typed, and may have associated constraints. Attribute constraints may serve not only to restrict possible attribute values as an extension to the typing scheme, but may also embody relationships between attributes and be capable of generating messages to other objects. Constraints applied to object attributes may be used as a powerful mechanism for inferring new attribute values from accumulated data and for propagating knowledge across the system. In this respect, the object attribute constraint mechanism is similar to the idea of procedural attachment to frame slots [Minsky 1975].

The current system is implemented in Quintus Prolog using object-oriented extensions devised by one of our collaborators in this project. C is used for programming the user interaction on a Sun 3 workstation.

3. The ISM user interface

A logically distributed system architecture with intelligent agents such as that of the ISM is ill-served by an imperative style of user interface where the user specifies exactly how the system should carry out some task. Rather, we wished to devise a declarative interface whereby the user specifies what task should be done and, as far as possible, the system should decide the best way in which task should be carried out.

There are two principal components to the interface system which we have devised. These are:

(a) A mailer agent which accepts all user inputs, analyses these inputs and directs messages to appropriate agents.

(b) An object browser agent which allows the user to examine and edit the contents of the object store directly.

These components work in tandem whereby messages are composed as objects using the browser system and passed to the mailer agent for processing and transmission. The architecture of this system is illustrated in Figure 2.

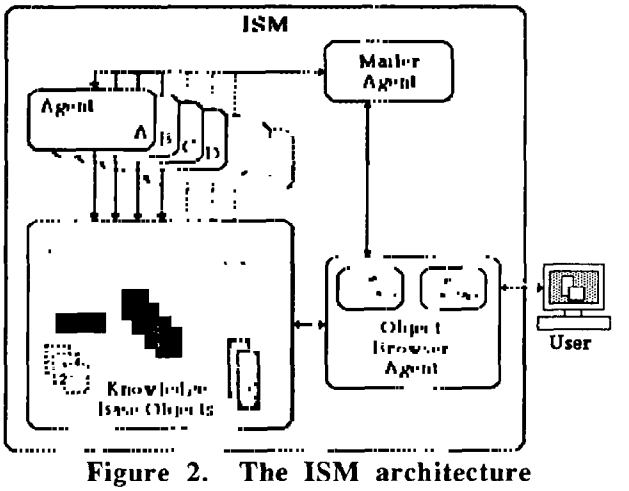

Figure 2. The ISM architecture

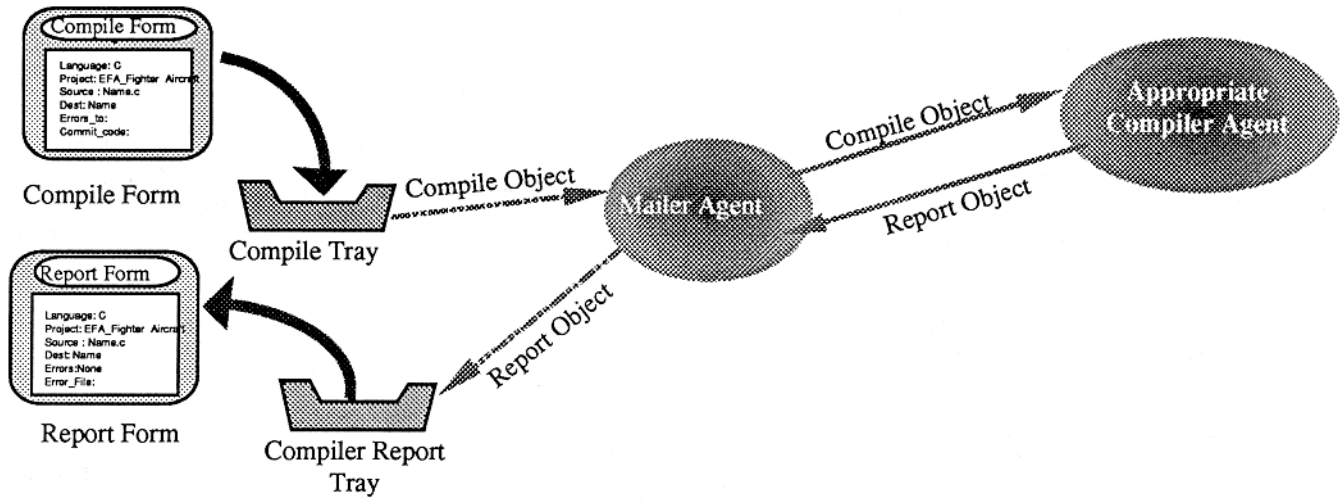

Figure 3. Interaction via the Mailer Agent

3.1 The mailer agent

The view of a system as a set of autonomous, interacting, intelligent agents is similar to that perceived by users of electronic mail systems [Hiltz 1981]. Thus, given that there are increasing numbers of people who regularly use electronic mail, we decided that an appropriate interface metaphor was that of an electronic mail system.

However, a characteristic of mail systems is the plethora of messages which must be processed by users [Hewitt 1977, Malone 1987] and this problem is likely to be amplified in a system where messages are generated automatically. The mailer agent provides the user with a range of facilities to construct a personalised interface to the system which can filter incoming mail messages. Thus, as well as co-ordinating communications between active agents, the mailer acts as an intelligent electronic mail system.

By adopting the mail system model agents may have a consistent interaction interface where exactly the same methods are used to send mail to system users, to file mail received and to initiate system actions. The conceptual view utilised by the mailer is the filling in and 'sending' of forms to other agents. An agent initiates any action within the ISM by passing an object to the appropriate agent via the mail system.

For example, to compile a program an agent (which need not be human) fills in a COMPILE form (object) this form is then submitted to the mailer agent which *'mails'* the completed COMPILE form to the appropriate compilation agent (Figure 3).

The advantage of this approach is that users need not know about the way in which the compiler is initiated or where the compiler is available. For example, say a distributed system dedicated one node as an Ada compiler server. Messages to compile Ada programs would automatically be directed at that node for procesing. The location of the node might change depending on overall system loading but this change would be completely transparent to users of the Ada compilation system.

3.2 Mailtray Structuring

Considerable research has been directed toward the development of efficient and reliable techniques for the transfer of messages but until recently comparatively little work has been done on interaction with Message Handling Systems[Malone 1987, Hutchison 1986]. Message Handling Systems currently tackle many of the problems generated by communication within collaborative environments by providing an efficient and reliable communication medium. However, due to the historical emphasis on the transfer of messages, and the subsequent lack of development of user interface techniques, the presentation of information to the user tends to be rudimentary and completely unstructured, generating an effect which has been termed *Information Overload* [Hiltz 1985].

When information overload occurs the flow of information is so rapid it makes it difficult for users to utilise a large amount of the information relevant to their needs. To be effective systems must give message recipients the ability to discriminate between those messages they wish to read and those of little relevance to them [Hiltz 1985]. Malone [Brobst 1986] conducted several studies of how various kinds of information are shared in organisations. He outlines a number of approaches people use when reducing the information they need to process.

The most interesting of the filter approaches he describes are cognitive filtering, where the decisions are based on the topic of the message, and social filtering, where decisions are based on who supplied the information. Additional studies[Sumner 1986] have shown that the majority of messages within electronic message systems are organizational, a significant amount of these are routine in nature, we believe this will also be the case within the ISM.

Our structuring technique is based upon the notion of *Mailtrays* which messages (forms) flow into and out of [Rodden 1988]. Each mailtray has a title, a number of attributes, and an action list which describes how forms should be processed. Additionally, each mailtray has an associated guard list controlling the nature of the forms held in the mailtray (Figure 4).

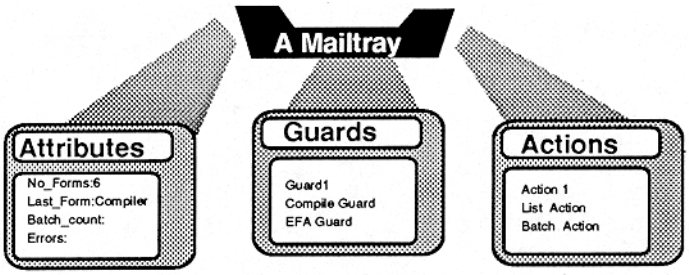

Figure 4. The Components of a Mailtray

Mailtrays are active elements which accept or reject forms depending on their guard list. A tray's guard list is composed by the agent to which it belongs and describes the criteria for adding forms to the tray. The agent is free to create an interface reflecting its particular classification of forms, which may be fine tuned as required. Mailtrays are dynamic so that an agent can amend this interface as requirements change.

Consider a member of a project team (a human agent) developing a component which interacts with various components developed by other team members. He or she may wish to arrange the various forms used so that all forms concerning errors regarding code of interest to him are grouped. It is also likely that he or she may wish to collate all communication with the compiler agents.

The team member can define a number of trays in order that forms of relevance are automatically sorted. Forms requiring immediate attention are dealt with directly, while routine forms are processed in as automated a manner as possible. Trays are defined by their creating agent using guards which define what messages may enter a tray, and an associated action list which describes what should be done with the forms placed in a particular tray.

The decision whether or not a form is placed in a mailtray is controlled by the tray's *guardlist*. The guardlist is a list of predicates which can be applied to the attributes of a form. If all the tests on a form's attributes succeed the form passes that guard and is added to a trays contents. Any number of guards may be associated with a mailtray and a form is accepted if any of these guards is true.

The guard list for the tray defined to collate all forms regarding compilation might simply be:

Guard *all* **class** *compile* ;

All forms belonging to the class *compile* may enter the tray but no others are accepted. However if the user wished to collect compilation forms from a particular project, say EFA_Flight_Simulator then he could alter the attribute list to allow only forms regarding EFA_Flight_Simulator to be added to tray by replacing the above guard by one of the form :-

```
Guard all class compile
    [
    project = "EFA_Flight_Simulator"
    ];
```

For a form to successfully negotiate this guard it must be of class *compile* and have an attribute called project with a value "EFA_Flight_Simulator".

When a form is placed in a Mailtray the *action list* for that tray is interpreted. Action lists consist of a list of commands to be executed. Each command is of the form:-

Action Head -> [Action Body];

The action head consists of the action name followed by the class of forms to which the action applies. If the form is of the class appearing in the action head then the succeeding action body is executed. Each action body is written in a notation consisting of IF THEN and assignment statements in conjunction with form handling primitives (mail, save, forward etc.).

For example a user may wish to process forms of class *code_test* only when at least ten forms of this type have been received. The action list for the appropriate tray might contain the action *batch* :-

```
batch : code_test  ->
 [
no_forms = no_forms +1;
IF no_forms >= 10
   THEN
          [
       Flag
          ]
];
```

No_forms is a user defined attribute associated with the tray to which this action belongs. Flag is a form handling primitive which informs the mailtray owner that the tray requires attention.

The mailtray user interface (Figure 5) utilises an icon based representation to allow the easy examination and manipulation of mailtrays. The interface uses direct manipulation of icons via a mouse in addition to keyboard input. A user wishing to interact with the ISM selects an appropriate form using the mouse, completes the form as required, and passes the form to the system by *sending* the form via the appropriate tray. Similarly all responses from the

system are placed in the appropriate trays using each tray's guard definitions.

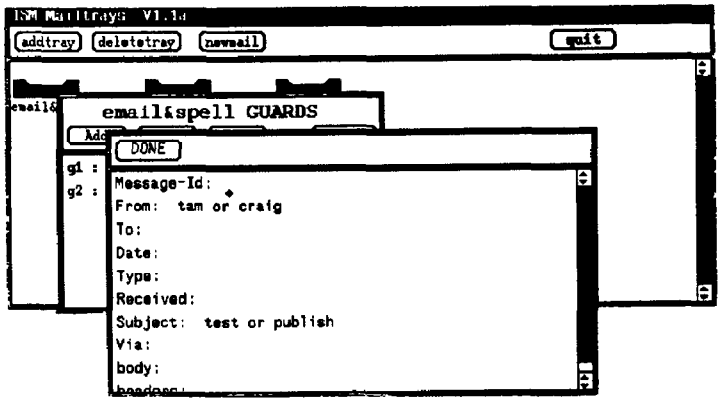

Figure 5. Mailtray Interface

3. 3 The object browser agent

The object browser allows users to interact with the system by directly manipulating [Schneiderman 1982] objects within the knowledge base. The system uses a *dynamic forms* metaphor based on a mapping of objects within the knowledge base to standard format windows (forms) on a workstation screen. The term *dynamic form* is used because users can alter the organisation of forms and can provide field relationships so that information may be added automatically to the form.

Dynamic forms are used to create instances of object classes, view existing instances, and define new classes. The mechanism is flexible enough to encompass a diversity of object classes, from passive objects; such as files of source code, to classes designed specifically to provide user control of the system; such as forms for sending electronic mail to other users.

Form based user interfaces are not a new idea [Smith 1983], and have been used in data processing systems for many years. Recent research has been devoted to using them as front-ends to systems running on machines equipped with bit-mapped displays and pointing devices. Cousin-Spice [Hayes 1983] is an example of such a system in which information in a form is structured into two types of field. These embody information representing commands, and that representing parameters. In object-oriented terms, these are analogous to methods and attributes respectively. The former type of field (a *button*) is *selected* by the pointing device to execute the command (*pressing* the button), while the latter (a *field*) requires a *value* to be associated with it by typing into a space adjacent to the field's label.

Dynamic forms are the physical representation of objects within the knowledge base. Thus, a user filling in a form, is actually creating or modifying an object. Any object in the knowledge base can be viewed as a form. Moreover, the powerful concept of constraining attribute relationships enables automatic assistance to be provided to the user. Thus, as a user assigns values to form fields, the system is able to make inferences about the values of associated attributes, both within and across object boundaries. Inferred values may be propagated across attributes as if the form were a spreadsheet.

The approach which we have taken is similar to what, in the *Information Lens System* [Malone 1986], are called semi-structured messages. It is observed that the structure imposed by the forms, as a framework, can encapsulate much information in fields which would otherwise have to extracted by parsing of free text.

The principal requirement of the object browser is that users must be able to inspect and manipulate the contents of the knowledge base. 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.

Two components implement the object browser agent (Figure 2):

- The *user interface*. This agent is responsible for physically presenting objects on the screen, handling user input, mouse clicks, etc. It performs the mapping of objects into forms. This is implemented as a set of objects and is reconfigurable and tailorable to individual user requirements.

- An *object library*. This is essentially a server to the user interface, it forms the object browser's interface to the mailer agent and hence to the rest of the IPSE. Requests from the user for information about some object or group of objects is relayed by the user interface to the object library. The object library initiates searches of the knowledge base, collects information about the required object(s) and returns the object data to the user.

The mailer agent collaborates with the object library to co-ordinate searching of the knowledge base and for the delivery of objects to appropriate agents on completion of their manipulation by the users.

3.3.2 A browsing example

Browsing the object store may be done on two levels; a definition level and an instance level. These correspond to looking at what object classes exist, and to viewing instances of a particular class, respectively. At the instance level users can view individual objects as completed forms, and create new instances by filling in a blank form.

Figure 6 shows browsing at the instance level. An object class *view_instances* is provided by the system to support the viewing of instances of other object classes.

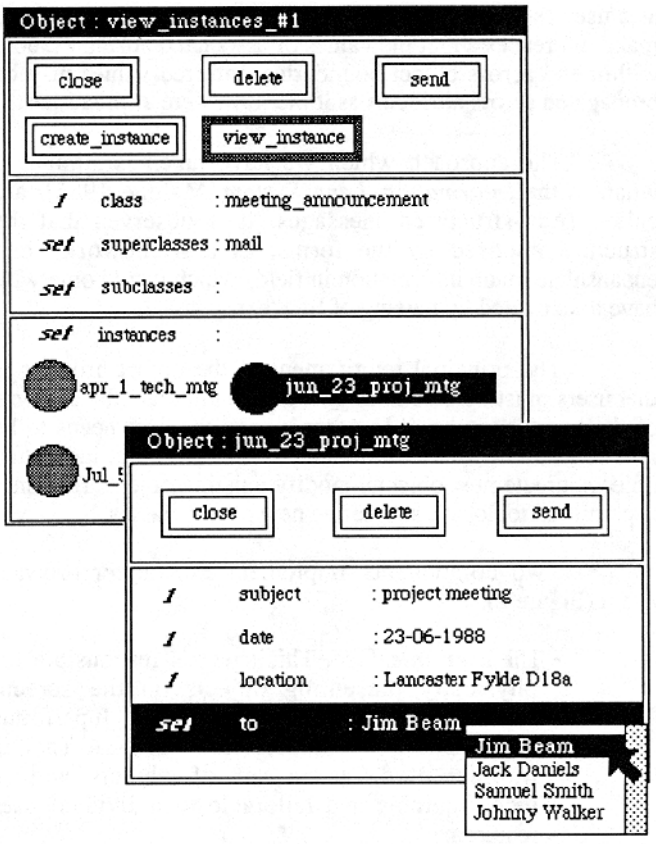

Figure 6. Viewing an instance of the class meeting_announcement

The instance of view_instances is represented by the overlaid form in the top left of the figure, labeled "Object : view_instances_#1". The form consists of a window containing three subwindows:

- A control panel containing buttons representing the messages defined for the class view_instances. The methods *close, delete* and *send* which are generic to all objects.
  Close removes the form from the display, *delete* removes the form representing the object from the display and deletes the object from the knowledge base, releasing the space which it occupied and *send* removes the form from the display and dispatches the object to the mailer for appropriate processing by other agents.
  Create_instance and *view_instance* are described below.

- The middle subwindow contains three fields; *class, superclasses* and *subclasses*, corresponding to attributes of the same name belonging to the class *view_instances*. The figure *1* and the words *set* adjacent to the attributes indicates their type. Currently available attribute types are; unique (1), set, bag and bigtext (used for unstructured attributes - source code, mail text, etc).

- The bottom subwindow contains a set type field; *instances*. A constraint associated with the attribute causes multiple values to be displayed simultaneously in iconic form.

In the example, a user has assigned a value to the *class* attribute, indicating that he or she wishes to inspect instances of the class *meeting_announcement* . A constraint associated with the *class* attribute enables values of the other three attributes to be automatically inferred. The following values have been generated:

- The attribute *superclasses*, has been assigned the value *mail* which represents the most immediate superclass.

- A *null* value has been generated for the *subclasses* attribute indicating that the *meeting_announcement* class represents a leaf of its inheritance tree.

- A set of three values for the *instances* attribute have been generated, *apr_1_tech_mtg, jul_5_tech_mtg,* and *jun_23_proj_mtg*. These represent instances of the class *meeting_announcement* resident within the knowledge base.

In the example, the user has opted to inspect the *jun_23_proj_mtg* object. The object library has responded by searching the object store for the required object and returning its details to the object browser. The object browser has mapped these onto the form seen in the bottom right hand part of the figure. This form encloses two subwindows.

- The topmost subwindow contains the generic message buttons; *close, delete,* and *send.*

- The lower subwindow contains the object's attributes and associated values.

The example illustrates the alternative scheme to viewing *set* and *bag* type attributes as icons (c.f. the *instances* attribute associated with *view_instances_#1*). The user has selected the *set* type attribute *to* and the object browser has caused a scrollable pop-up menu to appear listing all the attribute values.

A user may attempt to change an object by editing one or more attribute values. Whether this is permitted 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 and have protected attribute values. Such a case would spawn a warning message object, displayed (again, using our dynamic forms metaphor) as a form. For other classes of object it may be appropriate to allow objects to be modified.

Figure 6 illustrated the use of the *view_instances* object class for viewing existing objects. The *view_instances* form may also be used to create new instances of a class. The

create_instance method causes a blank form representing the class *meeting_announcement* to be displayed. An additional field, *name* is displayed to which the user is required to assign a value in order that the object may be uniquely identified. On completion of the form the *send* button is pressed. The object library creates the object (mapping the form contents onto the class definition) which is then dispatched to the mailer agent for processing.

4. Conclusions

The design of the next generation of Integrated Project Support Environments will differ greatly than that of previous generations. The evolutionary design which has led to the development of current IPSEs from programming environments is not capable of accommodating the emerging technologies which simplify automation of the whole software process.

The system described in this paper explores the potential for exploiting knowledge based programming techniques for the attainment of a high degree of integration within a support environment. It is postulated that a federation of intelligent, co-operating agents form an appropriate architecture for future environments. The prototype environment described here has been designed using an open systems philosophy such that new intelligent agents can be added incrementally as technology enables their production. Thus tasks which may currently require a human to perform, may be devolved to computer agents in the future. The first demonstrator version of this system was completed in July 1988 and evaluation is now underway.

We believe that an essential prerequisite to achieving a sufficiently high degree of integration and co-operation within our environment is the use of a consistent communication mechanism among agents and users. An intelligent mailer agent has been designed to enable the kind of group communications which must take place in such an environment. A tailorable user interface to the communication facilities permits users to interact with the system in such a way that its internal details are made transparent.

Acknowledgements

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.

References

[Alderson 1985] Alderson A., Bott M.F., Falla,M.E.: 'An overview of the ECLIPSE project'. In Integrated project support environments (John McDermid, Editor), London: Peter Pergrinus, 1985.

[Brobst 1986] Brobst S. A., MaloneT., et al.: 'Toward intelligent message routing systems' In:, R Uhlig (Ed.), Computer Message systems -85. Proc. 2nd International Symposium on Computer Message Systems. North Holland, Amsterdam, 1986.

[Campbell 1986] Campbell R.H. : ' SAGA: a project to automate the management of software prodution systems' in Software Engineering Environments (I.Sommerville Ed),London: Peter Pergrinus,1986.

[Curtis 1987] Curtis B, Krasner H, et al: ' On building Software Process Models Under the Lampost', Proc. 9th International Conference on Software Engineering.

[Dowson 1987] Dowson M.: 'Integrated Project Support With ISTAR' . IEEE Software, Vol 4. No 6, November 1987, pp 6-16.

[Goldberg 1983] Goldberg A. Robson D.: 'Smalltalk-80 The Language and it's implementation' Addison - Wesley, 1983.

[Hayes 1983] Hayes P.J., Szekely P.A.: 'Graceful interaction through the COUSIN command interface'. Int. J. Man-Machine Studies 19, 1983.

[Hewitt 1977] Hewitt C.: 'Viewing control structures as patterns of passing messages'. Artificial Intelligence, Vol. 8, no. 3, 1977.

[Hiltz 1981] Hiltz S.R and Turoff M. : 'The evolution of user behaviour in a computerized conferencing system', Comm. ACM, Vol 24, No 11 , November 1981, pp 739-752.

[Hiltz 1985] Hiltz S.R , Turoff M.: 'Structuring computer-mediated communication systems to avoid information overload', Comm ACM, Vol 28, No 7, July 1985.

[Hutchison 1986] Hutchison D., Armitage R., Muir S.J.:' A User Agent for the Unix Mail system'., Data Processing Vol 28, No 10, Dec 1986.

[Malone 1986] Malone T.W., Grant K.R., Turbak F.A.: 'The Information Lens: An Intelligent System for Information Sharing in Organisations'. Proc. CHI'86 Proceedings, 1986.

[Malone 1987] Malone T.W., Grant K.R., Turbak F.A., Brobst S.A., Cohen M.D.: 'Intelligent Information Sharing Systems'. Comm. ACM, Vol. 30, No. 5, 1987.

[Minsky 1975] Minsky M.: 'A Framework for representing Knowledge'. The Psychology of Computer Vision (P. Winston, Editor), McGraw-Hill, 1975.

[Rentsch 1980] Rentsch T.: 'Object Oriented Programming'. SIGPLAN Notices, OOPS 80, 1980.

[Ritchie 1978] Ritchie D. M., Thomson K. : 'The UNIX ime sharing system', Bell Systems Technical Journal, 57(6), 1905-29.

[Rodden 1988] Rodden T., Sommerville I. 'Mailtrays: An Object Orientated Approach to Message Handling ', Proc. EURINFO 88, First European Conference on Information Technology for Organisational Systems, Athens 16-20 May 1988.

[Schneiderman 1982] Schneiderman B. : 'The future of interactive systems and the emergence of direct manipulation'. Behaviour and information technology, Vol. 1, No. 3, 1982.

[Smith 1983] Smith C.D., Irby C., Kimball R., Verplank W., Harslem E. :'Designing the Star User Interface'. In Integrated Interactive Computing Systems (P. Degan and E. Sandewall , Editors), North-Holland, 1983.

[Sumner 1986] Sumner M.:'A Workstation Case Study', Datamation, Feb 15 1986, pp71 - 79

[Teitelman 1985] Teitelman W. : 'A Tour Through Cedar', IEEE Transactions on Software Engineering, Vol SE-11, No 3, March 1985.

[Wegner 1987] Wegner P.: 'Dimensions of Object-Based Language Design'.Proc. OOPSLA'87 , ACM press, 1987.

[Winograd 1973] Winograd T.: 'Breaking the Complexity Barrier (Again)'. Proc. ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages-Information Retrieval,Gaithersburg, Maryland, 1973.

[Zdonik 1985] Zdonik S.B. , Wegner P. : 'A Database Approach to Languages, Libraries and Enivironments," Proc. Workshop on Software Engineering Environments for Programming-in-the-Large, June, 1985