6. Object-oriented design for excavation simulation programming

Leonor Barroca
Department of Computer Science, University, York YO1 5DD, U.K.
email:lmb@uk.ac.york.minster
Sebastian Rahtz
ArchaeoInformatica, 5, Granary Court, St Andrewgate, York YO1 2JR, U.K.

6.1 Introduction

It is relatively unfashionable either to discuss the details of programming languages for archaeology, or even to imply that the average archaeologist needs to have any concept of what a programming language is. On the one hand, it is argued, any program can be written in any language, and arguments about which language to use are the province of the theoretical computer scientist; on the other hand, there are few applications which still need 'low-level' programming. The proceedings of the *Computer Applications in Archaeology* conference seem for some years to have eschewed even the discussion of database technology which was such a feature of earlier volumes. Why resurrect the language debate in the 1990s?

We would argue against both of the premises outlined above about programming:

1. The history of programming (Brooks 1982) has proved over and over again that only a small proportion of eventual effort has been spent in initial program coding; traditional projects have invested far more in maintenance. There are two solutions to this wasted effort: a complete and reliable system for the formal specification of programs, so that maintenance is minimized; and a programming environment which models the problem in a manner intuitive enough for the maintainer to be able to address conceptual problems rather than implementation bugs. Archaeologists involved in the creation of programs have to be part of the conceptual process.

2. The traditional contrast between 'programming' and 'applications' is dying away, simply because the increased sophistication of applications now provide a complete programming environment (a computing system built on an application which itself sits over an operating system on a 'real' computer). The most obvious example of this is the very widely used Hypercard software on Macintosh computers (mimicked by more recent programs like Supercard, Plus and Toolbook). Hypercard is rightly regarded as 'application' software, functioning like a free form database in its simplest mode, but it can be extended in almost any direction, leading the user seamlessly via form creation and automated production of links, to structured scripts and even new functions coded in a conventional language. The same applies in a lesser degree to the more sophisticated spreadsheets like Excel, and word-processors like Word, which have very full programming languages attached.

A large proportion of computer users 'indulge' in programming; just as the cheap camera lowered the standards of photography, just as instant food damaged careful food, and just as typewriters savaged typesetting (and as desktop publishing has recently gone over the corpse for a new assault), we may inevitably be in danger of more and more poor programming work taking place.

This paper considers whether the choice of programming paradigm can materially affect the immediate project (in this case, an excavation simulation system), or the stable maintenance of a product. We will contrast three implementations of the same basic system, and try to assess which is most likely to succeed in the long-term. We will first describe the idea of *object-oriented* programming; then outline the scope of the problem and the current implementations; then deal with an object-oriented approach using Smalltalk; and finally try to assess which of the implementation approaches is most successful.

6.2 Object-Oriented languages

6.2a A bit of history

The 'invention' of structured programming in the 1960s stands as a milestone which has very strongly influenced all of the developments that have taken place since. From the idea of *structure*, the concept of *modularization* evolved and with it ideas of *encapsulation* and ultimately of *object*.

The first language that introduced the main concepts of object-orientation (Stefik & Bobrow 1986; Korson & McGregor 1990; Wegner 1990), *object*, *class*, and *inheritance*, was SIMULA (Dahl *et al.* 1970), a language written mainly for simulation and still in use today. Objects are entities with a state and a set of operations that alter that state. The state is hidden and the set of operations determine the messages to which the object can respond. Classes are descriptions of similar objects which serve as templates for the creation of new objects. Inheritance is the principle behind reusability — new classes can be defined based on classes which have already been defined.

Smalltalk (Krasner 1983) was derived from a sequence of experimental languages in the 70s by Xerox, and became popular during the 80s partly because it combined a fundamentally new concept of programming with a complete system interface. The design of the latter has had an even wider effect than the language, as it is the origin of most of the windowing systems which are the desktop norm in the 1990s (the Macintosh system, Microsoft Windows, GEM and the X Window system are the obvious

examples; the system which popularized the interface, the Macintosh, was very directly based on the Xerox work).

Since their instantiation in Smalltalk, the ideas behind object-orientation have been widely applied in simulation, systems programming, artificial intelligence, and electronic publishing. The general term *object-oriented* has become very popular and there has been a great deal of research integrating the concept of *object* into other paradigms such as functional and logic programming, concurrency, and also database languages. The development of the C language into the object-oriented C++ has provided a very widely used implementation of the concepts. We must also stress that object-oriented design of programs independently of the chosen implementation language is a vital modern technique. Smalltalk has been used in the research described in this paper, simply as an example, rather than as a prescription for object-oriented programming.

6.2b Object-Oriented databases

Database research in the 1980s has been heavily influenced by the object-oriented paradigm, and the concept of object has been extended to this field. Object-oriented databases (Baroody & DeWitt 1981; Bancilhon *et al.* 1988; Kim & Lochovsky 1989) maintain the main characteristics of traditional databases; but they also have features which make them attractive for many applications, such as the possibility of defining complex objects. An object has an internal state that can be seen as a set of fields, but these fields can be defined as complex structures (sets, lists, tuples, or even actions) or as other objects. Object-oriented databases provide a way of guaranteeing the persistence of data; objects are maintained from one access of the database to the next, and they can be shared between different users and application programs as in traditional databases. Security and reliability of the data are important characteristics of databases that are also respected by the object model. A discussion of object-oriented databases is given in Barroca (1990).

There are an increasing number of applications that have requirements which are not easily adjustable to traditional database schemas; there are various reasons for this:

- the *changing* complexity of data, which is still difficult to deal with in the relational model;
- the need for new data types (perhaps the biggest single problem with most database systems);
- the need to assign and interpret a complex meaning to data, and the implications for updating;
- the problem of integrating the database system with conventional programming, this remains a difficulty.

It is in fields where such needs are felt that object-oriented databases have found important applications; although the ideas apply in many areas, databases of spatial information with complex structures, and image databases, have been promising fields. These are, of course, at the heart of archaeology.

6.2c Smalltalk

Smalltalk-80 was developed in Xerox Palo Alto Research Centre (Goldberg & Robson 1983; Goldberg 1983; Krasner 1983). It consists of a graphical, interactive, programming environment which allows the programmer to think in terms of *objects*. An object in Smalltalk consists of private memory, the data, and a set of operations, called *methods*, that it can perform; every object has an identity. The other main component of the system, a *message*, is a request directed to an object asking it to perform some action. It is up to the object that receives it, the *receiver*, to decide how the action will be performed — the *method* to be invoked. The set of messages that an object is prepared to answer is called its *interface*.

Every object is an *instance* of a *class*. All the instances of a class have the same message interface and they all use the same set of methods. Each instance has its own set of instance variables (private data) and they share class variables. The class defines the behaviour and the attributes of the instances, and also how new instances are created. A class can inherit from another class (the superclass) and specify new attributes and behaviour; this new class is called a subclass. Therefore a hierarchy of classes can be defined; the Smalltalk-80 system provides a set of classes with a basic functionality.

Smalltalk/V (Digitalk Inc. 1986) is a reduced version of Smalltalk-80 that has been used for this implementation. It is available for an IBM-PC compatible running PC-DOS or MS-DOS, with at least 512K of RAM and a graphics card.

The Smalltalk system is menu-oriented and the interaction with the environment is done through the selection of options in the menus. Several windows can be opened; each window has its own menu. A window can be divided into different panes and there is a menu for each pane. Outside the windows the system menu allows the creation of new windows, and performing system functions. The Smalltalk interface was one of the first WIMP (Windows, Icons, Mouse, Pop-up Menus) interfaces with its panes, windows and menus.

In Fig. 6.1 we show a special window provided by the system, the *Class Hierarchy Browser*. This class allows the *browsing* of the classes already defined, editing the definitions and creating new ones. It is divided in three main panes: *class hierarchy list*, *method list* and *contents*. The *class hierarchy list* on the top left shows the hierarchy of the classes already defined (e.g. Sygraph is a subclass of FreeDrawing, and Digging is a subclass of Sygraph); creating a new class corresponds to create a subclass of one of the classes already defined. The *method list* on the top right shows the list of methods of the class selected (e.g. 'digging' is an instance method of Sygraph; note that it is entirely distinct from the class 'Digging'). Under this pane there are two small panes which allow the selection of either *instance* or *class* methods. New methods can be created selecting the corresponding option in the *method list* pane. The description of the method selected appears in the *contents* pane at the bottom of the window. This pane can be edited.

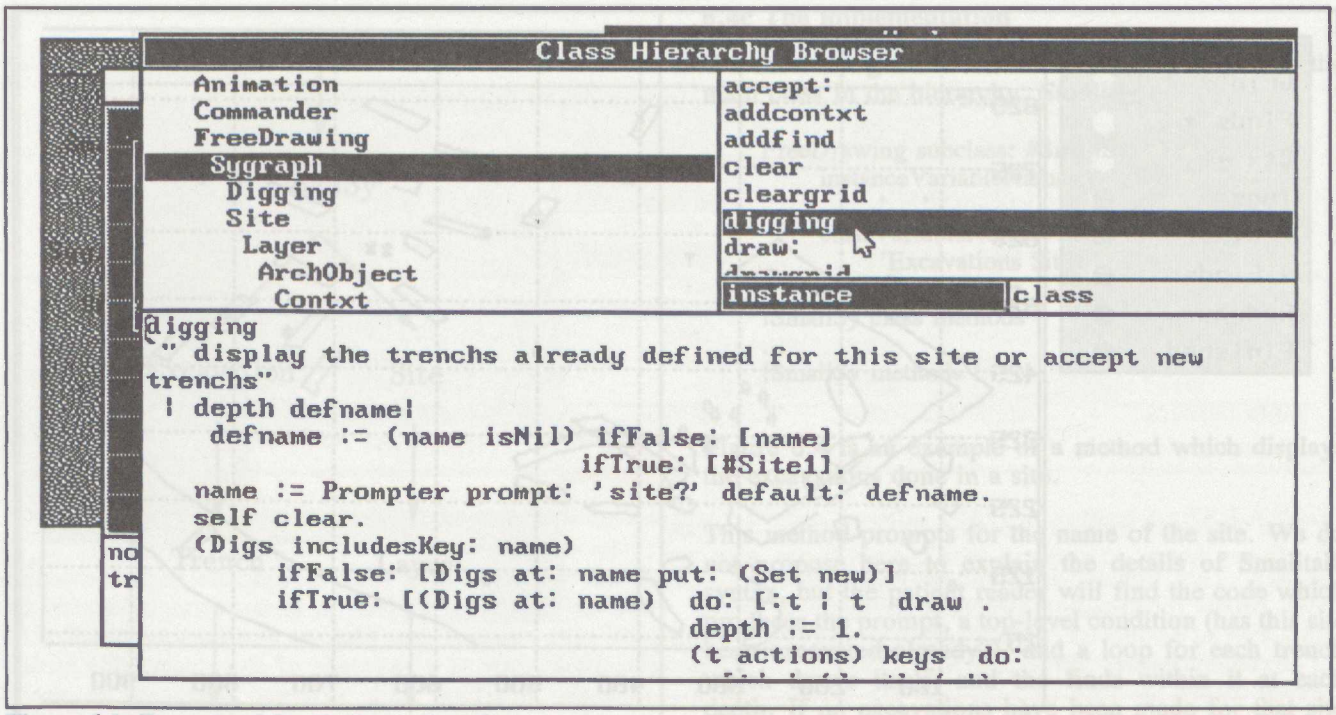

Figure 6.1: Example of Smalltalk user interface.

6.3 The SYASS project

The history and aims of the SYASS project have been covered in some detail in other papers (Rahtz 1988; O'Flaherty *et al.* 1990; Wheatley 1991). The result was a resource-based graphical excavation simulation system written by Dave Wheatley, used for several years by students at Southampton and York; this was preceded by a simple text-based program (CemySyass), and two attempts have been made to rewrite the program (SMALLSY, discussed in this paper, and a version to run under Unix and the X Window graphical user interface, written by Mark French as a Computer Science undergraduate project at the University of Southampton). Each implementation consists of a database of material describing an archaeological site or area, and an interface which permits a user to 'buy' knowledge from the system (the equivalent of excavation) in order to try and find out what happened on the site.

Fig. 6.2 shows a typical screen in SYGRAF, a two-dimensional representation of an excavation in progress showing features that have been recovered.

6.4 The Smalltalk implementation of SYGRAF

6.4a Object-Oriented design methodology

Smalltalk has often been used as a tool for modelling and prototyping. We wanted to test its suitability in the initial phase of designing an archaeological project (in this case an excavation simulation) and rapidly creating a prototype. Subsequently we were able to develop the system to a point where we could compare it with other implementations.

The initial approach followed an object-oriented design methodology (Henderson-Sellers & Edwards 1990; Bailin 1989; Booch 1986) because it seemed appropriate for such a hierarchical and object-like structure as the archaeological excavation.

1. The problem is analyzed in terms of the objects that compose it and the services they provide.
2. Once the objects are identified, we can define classes which contain the attributes and the functional characteristics of each object.
3. Interactions between objects are established through the services they require and those that they provide.
4. At the same time the internal structure of the objects is established in a bottom-up way.
5. As a following step, we may decompose the classes into more detailed subclasses.

This is a process that has to be iterated as new objects are identified. We will end up with a well defined hierarchical structure that directly reflects the virtual world which we are trying to model. The structure gives priority to the functionality. This has the advantage that the algorithms and the internal data only become fixed at a later stage, giving to the whole system a greater flexibility, and making it easier to approach changes at either the top or the bottom with greater confidence.

6.4b The model

For the archaeological excavation simulation, the hierarchy has a clear structure, shown in Fig. 6.3. The distinction between archaeological activity on the left, and raw data on the right, is fundamental to the design of all the SYGRAF projects.

6.4b (i) The objects/classes

A SMALLSY simulation is defined as having a set of sites, each with a unique name and a set of layers.

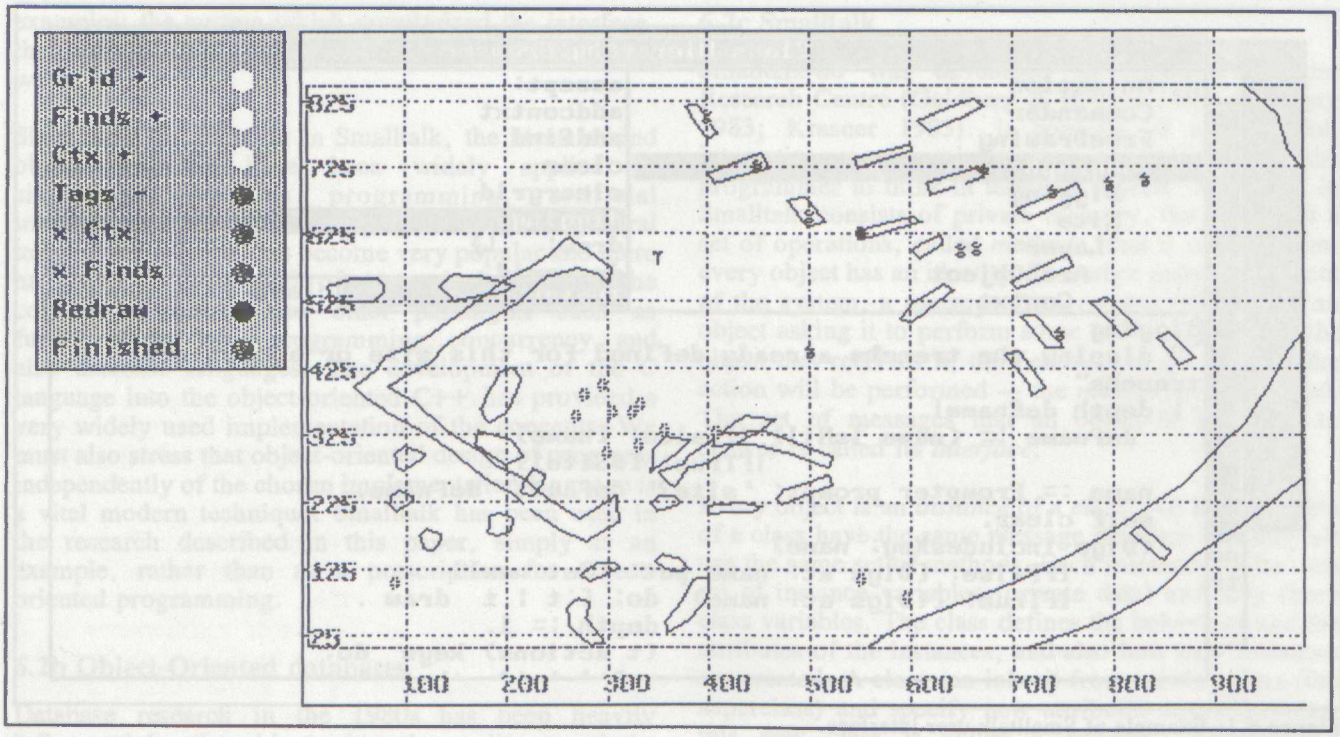

Figure 6.2: Typical SYGRAF screen in the middle of excavation.

Each layer has a depth, a set of contexts and a set of finds.

Excavations are viewed in terms of the trenches that have been dug in a certain site. For the first prototype, we have not considered different users making an excavation on the same site, but this is an extension that can be easily achieved.

Each trench is defined as a rectangle with information on the depth to which it has been excavated and the actions taken at each depth of digging. Each action contains the information relating to date, cost, tool used, etc.

The objects identified will induce the classes **SmallSy**, **Site**, **Layer**, **Find** and **Context** for the required static information. The simulation involves **Excavation**, **Trench** and **Action** classes.

6.4b (ii) Formalizing the Model

This description given above can be formalized, defining the structure of each object in terms of data types. We will see afterwards how closely the following definition will be related to the implementation.

SmallSy	=	**tuple** (Excavations, Sites)
	=	
Sites	=	**ff** (Name, **set Layer**)
Layer	=	Find **U** Context
Find	=	**tuple** (Point, Description)
Context	=	**tuple** (**set** (Point), Description)
	=	
Excavations	=	**ff** (Name, **set** (Trench))
Trench	=	**tuple** (Rectangle, **ff** (Depth, Action))
Action	=	**tuple** (Date, Cost, Tool, ...)

We have described above, in a formal way, the structure of each class. Each class corresponds to a type, and we build new types with constructors that operate on types. The constructors used above are the finite function, the union, the tuple and the set. A finite function (**ff**) constructs a new type from two types: the domain and the range. A finite function is a set of pairs whose first element (from the domain) is unique (a key). The union (**U**) builds a new type whose elements are members of one of the primitive types. The tuple (**tuple**) builds a new type whose elements have several components, each one corresponding to one of the primitive types. Finally, the set (**set**) builds a type whose elements are sets of elements of the primitive type.

6.4b (iii) The hierarchy

Defining the hierarchy of the classes can be sometimes a delicate question (Lieberherr & Riel 1989). We have to identify two main points: *commonality* and *specialization*. Commonality occurs when several classes have been identified and they share data and behaviour. This is the case of classes **Context** and **Find**. In this case an abstract class should be defined, **Archaeological Object**, from which the others will inherit data, the description and the behaviour related to it.

Specialization occurs when there is a class for which a lot of software has been written and we want to add some items creating a new class. This happens, for example when we define our main class **SmallSy** as a subclass of a class provided with the system, **FreeDrawing**. We take advantage of the drawing facilities already defined and adapt them for our simulation. Specialization needs to be carefully thought out because changes cannot be made to the previously defined class without affecting the new one.

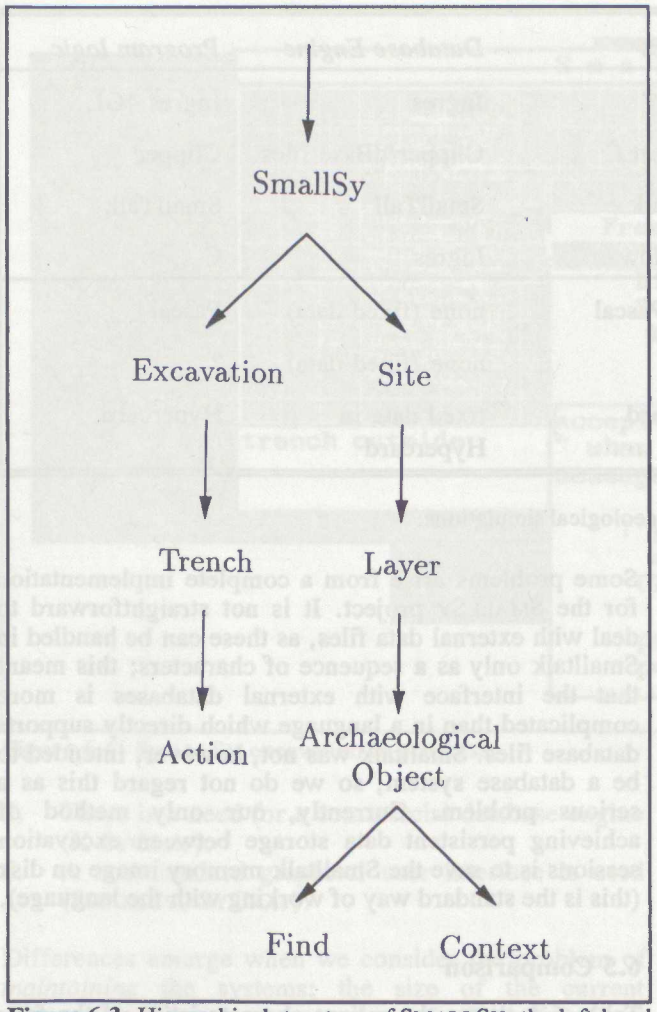

Figure 6.3: Hierarchical structure of SMALLSY; the left-hand side represents the archaeological activity, while the right-hand side represents the archaeological raw material. Note that the recording method which classes finds and contexts as entirely different objects was chosen arbitrarily, and would probably not be that used by the majority of excavators.

6.4c The implementation

We show the generic structure of the definition of the main class in the hierarchy, **SmallSy**.

```
FreeDrawing subclass: #SmallSy
        instanceVariableNames:
                '... '
        classVariableNames:
                'Excavations Sites '
                '... '
!SmallSy class methods !
...
!SmallSy methods !
...
```

Figure 6.4 is an example of a method which displays the excavations done in a site.

This method prompts for the name of the site. We do not propose here to explain the details of Smalltalk syntax, but the patient reader will find the code which produces the prompt, a top-level condition (has this site been excavated already?), and a loop for each trench which draws itself, and the finds within it at each depth. If no excavations have been made for that site yet, it creates an empty set of trenches for that site and the excavation is defined, so that new trenches can now be added. As we recall from the formal definition, an excavation is a pair with a name and a set of type **Trench**. If we check the formal definition, we see that a trench has a finite function of **Depth** and **Action**. We look for the highest depth and send a message, *showfinds*, to itself to show the finds inside that trench for a certain depth.

It is appropriate at this stage to refer briefly to the graphical concepts of Smalltalk and how they have been used here.

Smalltalk/V is based on *bitmapped* graphics. Every graphic entity is represented as a vector of dots. A dot is displayed as a pixel and is stored as a **Bitmap** contained in a **Form**. **Form** is the class that defines the objects that hold images. Images are displayed by transferring a form that contains the image onto the

Figure 6.4: Example method for displaying a site.

```
digging
"display the trenches already defined for this site or accept new trenches"
  | depth defname|
     defname := (name isNil) ifFalse: [name]
                             ifTrue: [#Site1].
  name := Prompter prompt: 'site?' default: defname.
  self clear.
  (Excavations includesKey: name)
     ifFalse: [Excavations at: name put: (Set new)]
     ifTrue: [(Excavations at: name) do: [:trench | trench draw.
                                          depth := 1.
                                          (trench actions) keys do:
                                          [:key | (depth < key)
                                          ifTrue: [depth := key]].
                                          depth to: 1 by: -1 do:
                                    [:d | self showfinds: trench at: d.]]].
  self newState: #accept:.!
```

Name	*Operating System*	*Graphics*	*Database Engine*	*Program logic*
CemySyass	Unix	none	Ingres	Ingres 4GL
SyGraf	MS-DOS	Microsoft C	Clipper/dBase files	Clipper
SmallSy	MS-DOS	SmallTalk	SmallTalk	SmallTalk
X-SyGraf	Unix	X Window	Ingres	C
Fugawiland	MS-DOS	Turbo Pascal	none (fixed data)	Pascal
Digging Deeper	MS-DOS	?	none (fixed data)	?
Santa Barbara	Apple Macintosh	Hypercard	fixed data in Hypercard	Hypercard

Table 6.1: Characteristics of SYASS software, and other archaeological simulations.

display. This transfer is achieved by an instance of class **BitBlt** (bit block transfer). Smalltalk/V defines a class **Pen** as a subclass of **BitBlt**; this class has a series of methods to draw with a pen; a pen has a form as its source where the size and tip size are defined. As a subclass of **BitBlt** it also has a destination form.

The definition in Fig. 6.5 is the method to draw a context that falls inside a trench. A context has an instance variable, *points* that contains a dictionary of points (each point has an order number). A pen is created, its colour is defined in a mask form. The pen is moved by the message *goto:*, and it is made to draw between all the points that are contained inside the trench. Note that there is no need for a special algorithm to design only the part of the context that is visible; this is easily done by defining the clipping rectangle of the pen to be the trench.

6.4d The SmallSy implementation

Creating an initial skeleton of SyGraf in Smalltalk took a few hours after the schema had been drafted on paper; there was no stage of translation into structured third generation language code. The model which solves the problem was drawn directly from the real situation (contexts are represented by a class of 'context', rather than by entries in a database and a set of procedures), and the implementation was directly obtained from this model. An example session is shown in Fig. 6.6.

Some problems arise from a complete implementation for the SmallSy project. It is not straightforward to deal with external data files, as these can be handled in Smalltalk only as a sequence of characters; this meant that the interface with external databases is more complicated than in a language which directly supports database files. Smalltalk was not, however, intended to be a database system, so we do not regard this as a serious problem. Currently, our only method of achieving persistent data storage between excavation sessions is to save the Smalltalk memory image on disk (this is the standard way of working with the language).

6.5 Comparison

Table 6.1 shows the salient characteristics of the four Syass implementations which have been attempted, and three other systems which perform a similar job.

None of these systems is ideal; apart from the fact that they all deal only with two dimensional data, they each suffer from at least one of the following problems in the delivered product:

1. The database is insufficiently separate from the program, so that one cannot 'plug in' different data (Fugawiland, Digging Deeper).
2. The database is written in a programming language, which has no convenient front-end for the casual user (SmallSy).
3. There is no graphical feedback (CemySyass).
4. Speed of operation is relatively slow (SyGraf).

Figure 6.5: Method employed to draw a context that falls inside a trench.

```
draw: aTrench colour: aDepth pane: pane
    "draw the points that are inside a trench. "
    | pen i rect |
      pen := Pen new.
      rect := pane frame.
    pen mask: (Form color: aDepth);
      clipRect: aTrench;
      place: (position scaleto: rect).
      i := 1 .
      [i <= points size]
        whileTrue: [temp:= (points at: i) scaleto: rect.
          pen goto: temp.
          i := i + 1.]
```

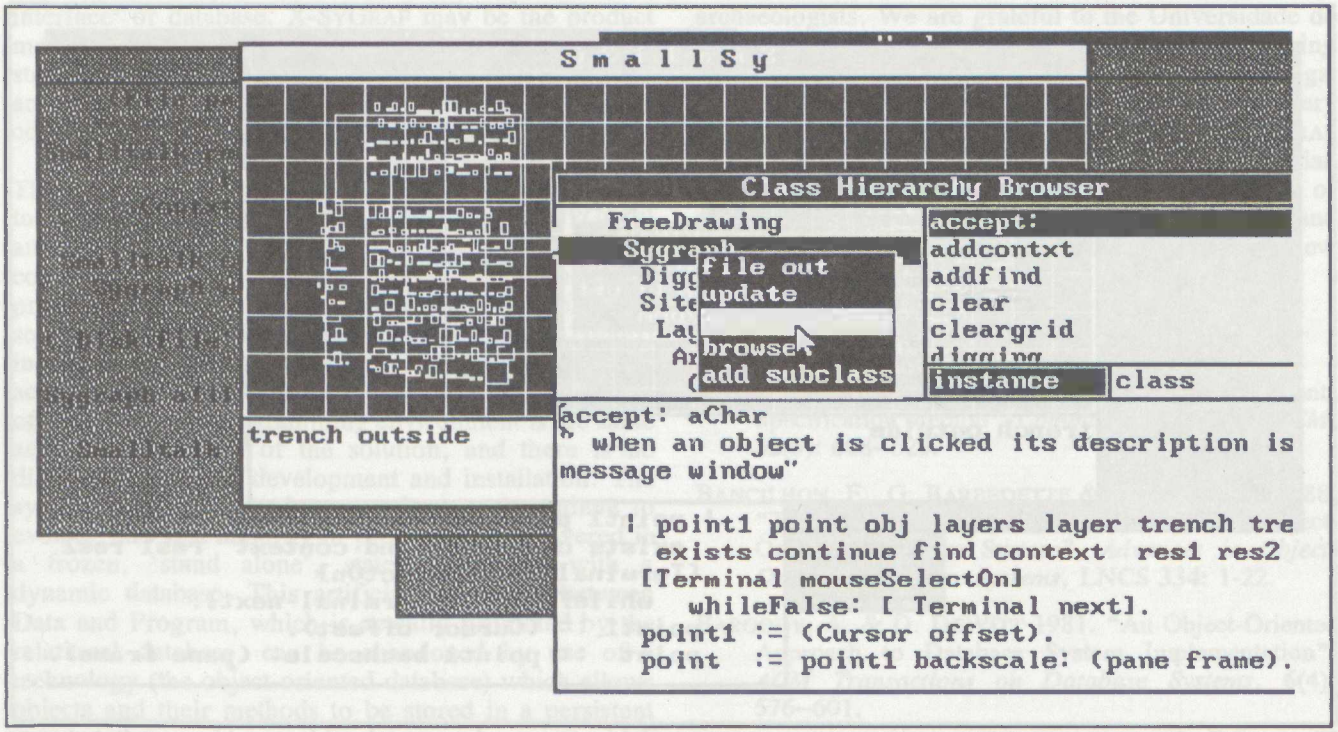

Figure 6.6: SMALLSY excavation in progress.

5. There is a need for a commercial database engine (X-SYGRAF).
6. A non-standard graphical user interface is used (SMALLSY, SYGRAF).

Differences emerge when we consider the problem of *maintaining* the systems; the size of the current program code is revealing, as shown in Table 6.2. Each of the programs depends on existing libraries for the graphics, and the first two depend on database management systems. Even the decidedly smaller size of the Smalltalk code conceals the fact that the environment offers a clearer view of the program (because the source is grouped by methods for classes), making it faster to locate problems.

The most likely problems arise in the area of program logic. In SYGRAF a hierarchical menu system is built in procedural code, involving a model of the *program* separate from the model of the *excavation*. In SMALLSY there is a single system of messages between objects, which involves fewer conceptual situations in which the programmer can go wrong. Similarly, in X-SYGRAF the C flow control is quite limited — the X Window system is event-driven, so it is simply necessary to link up the menu options and the event handlers to call C functions. Most of the time the system hangs around in a loop waiting for events to occur (XtMainLoop). The only code which is really not X or Ingres based is the code for handling the internal data structures (contexts, finds and trenches), and the code to work on the excavation, which then calls other separate functions to do the actual database work.

6.6 Conclusions

It is too early to talk about long-term maintenance of the Smalltalk version of SYGRAF, but the immediate project was a success. Apart from the fact that the design time was much shorter,[1] there was very fast development. A direct comparison was made between one author working on SMALLSY, and the other working on SYGRAF, on comparable computers; the incremental compilation and interpreted nature of Smalltalk meant that changes to the code took less than 1 minute to implement and test, whereas the traditional edit/compile/link/run cycle under MSDOS for SYGRAF took about 5 minutes.[2] The cycle for recompiling X-

Table 6.2: Comparison of developing the SYASS software.

	development	lines of code	result	notes
SYGRAF	6 months	5900 (Clipper) 2000 (C)	MS-DOS executable	time includes creation of site databases
X-SYGRAF	6 months	c. 3500	Unix executable	part-time only
SmallSy	4 weeks	800	SmallTalk image	no optimization for performance

1. Partly, of course, because we were building on the work done by Dave Wheatley.
2. Sceptics should note that use of the very slow Clipper *rtlink* linker was necessary for the project.

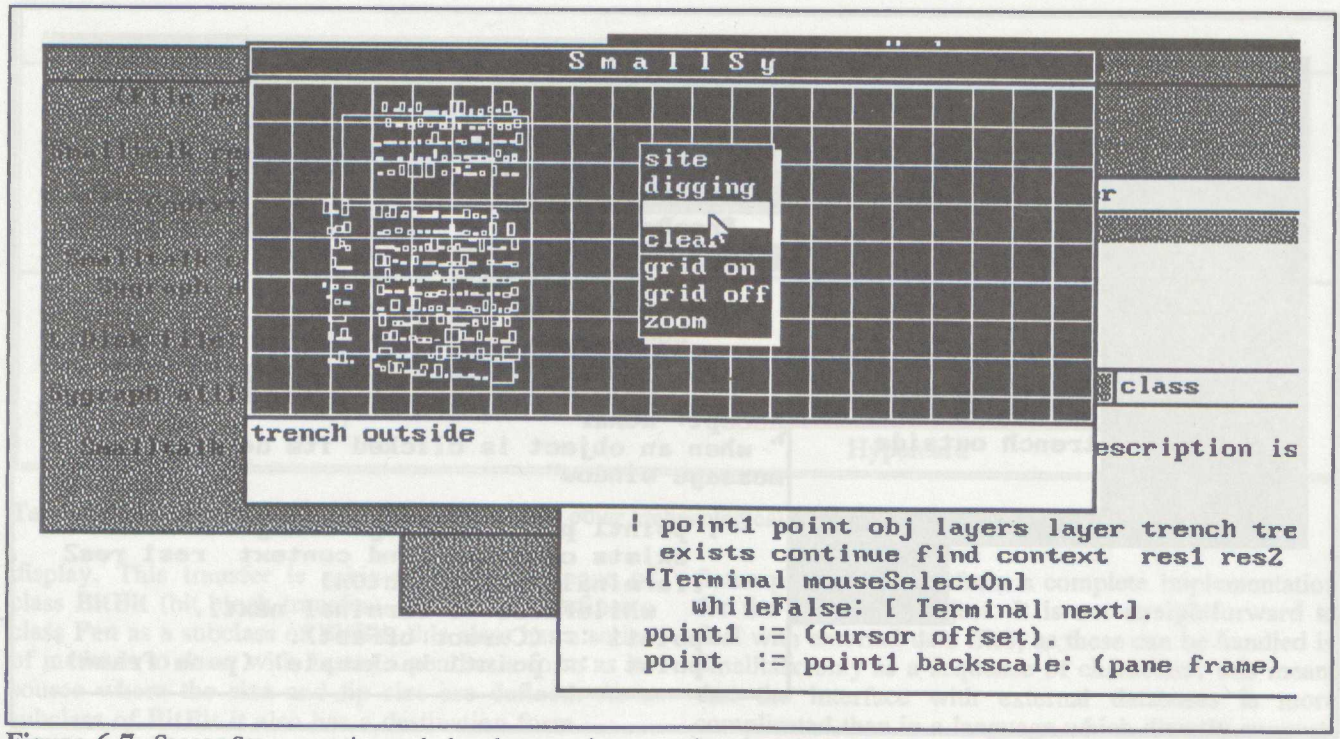

Figure 6.7: SMALLSY excavation and class browser in use at the same time.

SYGRAF on a Unix workstation varied from 48 seconds to over 5 minutes when many files were affected (the resulting executable file is 1.3Mb in size). There was even greater speed offered by Smalltalk, however, in dealing with conceptual problems; issues could be quickly related to the appropriate object in the system, and its methods inspected and changed. Moreover, SMALLSY is a system which can be extended at any time by adding new classes or methods, without the conceptual overhead of translating these into abstract procedures.

The next generation of SYASS software could well be implemented in Smalltalk. Its prototyping facilities which offered positive results from the start were encouraging. Using Smalltalk/V, we are tied to a proprietary product; SMALLSY only exists as a Smalltalk image, which cannot be distributed on its own. In fact, we are *always* tied to some proprietary software (such as MSDOS or Windows), but Smalltalk is insufficiently widespread to make it a good choice for an educational program. The Smalltalk user interface, while brilliantly ahead of its time when it came out, is sufficiently different from (say) Windows or Macintosh that users may have difficulty. We have lost the advantages of traditional database technology — enforcing of integrity, portability, re-useability, multi-user facilities, distributed data, etc. — by putting the data into Smalltalk objects, but we would hope that a final version would use an external database system for the main storage.

The advantages of developing in Smalltalk were clear:

1. We had a design of the system which we could use from the start.
2. The graphic capabilities of the system were easy to harness, and there was no need for much effort to be invested in graphics algorithms, and the concept of 'methods' for drawing objects allows us to change the method very quickly for a different implementation.
3. Creating new attributes of the objects or adding new behaviour were done in a very straightforward way without any need to change the rest.
4. The facility for continual testing and prototyping during the development of the program without any added effort resulted in few problems.
5. The main potential of the concept of object-orientation is its relation with simulation. Each real world object is represented by an object in the computer, and interaction between real world objects is modelled by message sending.

We should distinguish between the advantages of Smalltalk/V as a development environment, and the advantages of an object-oriented design and programming philosophy. We could have the advantages of object-oriented *design* by using a language like C++ (which involves a traditional edit/compile/link/run cycle), but the speed of *development* using Smalltalk comes from object-oriented programming (developing classes and methods, and running the program, in the same graphical system (Fig. 6.7). The dynamic binding of Smalltalk, which permits us to easily develop object browsers as part of the running system, is not an inherent part of object-oriented programming. It is likely that in future we would choose to use some system between C++ and pure Smalltalk, such as versions of Smalltalk which use Windows or Presentation Manager as their windowing system, or platform-independent libraries of C++ classes for the X Window, DOS or Macintosh systems.

The most fundamental immediate problem of both SYGRAF and SMALLSY, we suggest, is that neither of them uses currently accepted standards for the user

interface[3] or database. X-SyGraf may be the product most likely to succeed for this reason, since it uses standardized software; the display part can be taken out and rewritten without affecting the statements which could request data from any SQL-conformant database.

The difference between SyGraf and SmallSy relates to the distinction between the batch model (SyGraf) and the interactive model (SmallSy) of providing computer programs. SyGraf is written using a programming environment which exists to build solutions, which will be executed in another context independent of the development, and the installation is accordingly a closed process. In the interactive model of SmallSy, the programming environment is the same as the environment of the solution, and there is no distinction between development and installation. The system can be changed at any time, and continue to evolve. The vast majority of programs are delivered in a frozen, 'stand alone', state, interacting with a dynamic database. This artificial distinction between Data and Program, which is actually promoted by the relational database, can be abandoned by use of a technology (the object-oriented database) which allows objects and their methods to be stored in a persistent way together, and by working in an environment which supports the same objects as the program.

The SYASS project has effectively ended. Regardless of the success of the actual product, we believe that this type of mixed software is both possible and desirable, if we can find a middle road between conformance to standards, and *Metropolis*-like mechanisation of computer-based resources.

Archaeology must continue to develop a closer relationship between its data and its ideas by, at the least, using software technologies which maximize the importance of modelling their data. There are strong similarities between interpreting archaeology and object-oriented programming, both of which deal with a virtual reality which the user tries to relate to the real world. Both involve a continual search to identify objects (in the broadest sense), classify objects, and relate objects to one another. It does not necessarily follow that archaeological programs are best written using an object-oriented methodology, but if we are to make use of our large archaeological databases, we need to integrate them with our analytical tools. Many of our current databases are simply collections of dumb data, with no inherent structure to relate the tables to each other. To make use of them, archaeologists construct interpretations and models which refer back to the databases. We believe that it is time to explicitly link together our computerized resources into a coherent structured model, and we hope that this paper has demonstrated some of the advantages of an object-oriented paradigm in this endeavour.

Acknowledgments

This paper arises from a continuing collaboration between the authors on software development for archaeologists. We are grateful to the Universidade do Minho, Departamento de Informática, for allowing Sebastian Rahtz to take a short sabbatical in Portugal while working on the SyGraf project. We are very grateful to David Wheatley for the main SyGraf program on which our system is based; Brian Molyneaux commented on the paper from the point of view of a SyGraf teacher, and Mário Martins and Gillian Lovegrove commented from the point of view of computer science.

References

Bailin, S. 1989. "An Object-Oriented Requirements Specification Method", *Communications of the ACM*, 32(5): 608–623.

Bancilhon, F., G. Barbedette & V. Benzaken 1988. "The Design and Implementation of O2, an object-Oriented Database System", *Advances in Object-Oriented Database Systems*, LNCS 334: 1-22.

Baroody, A. & D. DeWitt 1981. "An Object-Oriented Approach to Database System Implementation", *ACM Transactions on Database Systems*, 6(4): 576–601.

Barroca, L. 1990. "Object-oriented database design in archaeology", *Science and Archaeology*, 32: 50–56.

Booch, G. 1986. "Object Oriented Development", *IEEE Transactions on Software Engineering*, SE-12(5): 211–221.

Brooks, F. 1982. *The Mythical Man-Month*, London, Addison-Wesley.

Dahl, O., B. Myhrhaug & K. Nygaard 1970. *Simula 67, Common Base Language*, Technical report, Norwegian Computing Centre.

Digitalk Inc. (ed.) 1986. *Smalltalk/V: Tutorial and Programming Handbook.*

Goldberg, A. 1983. *Smalltalk-80: The Interactive Programming Environment*, London, Addison-Wesley.

Goldberg, A. & D. Robson 1983. *Smalltalk-80: The Language and its Implementation*, London, Addison-Wesley.

Henderson-Sellers, B. & J. Edwards 1990. "The Object-Oriented Systems Life Cycle", *Communications of the ACM*, 33(9): 142–159.

Kim, W. & F. Lochovsky 1989. *Object-oriented Concepts, Databases, and Applications*, London, Addison-Wesley.

Korson, T. & D. McGregor 1990. "Understanding Object-Oriented: A Unifying Paradigm", *Communications of the ACM*, 33(9): 40–60.

Krasner, G. 1983. *Smalltalk-80: Bits of History, Words of Advise*, London, Addison-Wesley.

Lieberherr, K. & A. Riel 1989. "Contributions to Teaching Object-Oriented Design and Programming, in OOPSLA'89", *Proceedings of the ACM*

3. The fact was noted by Dave Wheatley as soon as SyGraf was started, but the immoderate learning curve for Windows forced the issue.

Conference on Object Oriented Programming, Systems, Languages and Applications.

O'FLAHERTY, B., S.P.Q. RAHTZ, J. RICHARDS, & S. SHENNAN 1990. "The Development of Computer-Based Resources for Teaching Archaeology", in S. Cacaly & G. Losfeld (eds.), *Sciences Historiques, Sciences du Passe et Nouvelles Technologies d'Information: bilan et evaluation*, Actes du Congres International de Lille (16-18 Mars 1989), CREDO, Universite de Gaulle, Lille III, Lille.

RAHTZ, S. 1988. "A resource-based archaeological simulation", in S. Rahtz (ed.), *Computer and Quantitative Methods in Archaeology 1988* British Archaeological Reports (International Series) **446**, Oxford, British Archaeological Reports: 473–490.

STEFIK, M. & D. BOBROW 1986. "Object-Oriented Programming: Themes and Variations", *The AI Magazine*, **6**(4): 40–62.

WEGNER, P. 1990. "Concepts and Paradigms of Object-Oriented Programming", *OOPS Messenger*: 7–87.

WHEATLEY, D. 1991. "SyGraf: resource based teaching with graphics", in K. Lockyear & S. Rahtz (eds.), *Computer Applications and Quantitative Methods in Archaeology 1990*, British Archaeological Reports (International Series) **565**, Oxford, Tempus Reparatum: 9–14.