Object Oriented Programming and Computerised Shared Material

Pål Sørgaard*
Computer Science Department
Aarhus University

Abstract

Computer supported cooperative work currently receives much attention. There are many aspects of cooperative work. One of these is the use of shared material. Much cooperation is based on silent coordination mediated by the shared material used in the work process. The properties of the shared material are, however, often ignored when work is computerised. Instead the emphasis has been on automating frequent work procedures. This has resulted in very inflexible systems.

A fundamental idea in object oriented programming is to model the phenomena in the part of reality the system addresses. These modelling techniques can be used to implement shared material on computers. The result is a raw system providing the material and the essential primitive operations on this material. Such a system can be seen as a specialised programming environment which can be tailored to the needs of individual users or be modified for future needs.

This use of object oriented programming requires persistent and shared objects. Some objects may be active and execute as parallel processes. Incremental change of a running system will be needed to allow evolution.

It is non-trivial to decide which properties of the material to model. An example demonstrates that this decision may depend on the kind of technology being considered.

Key words and phrases: computer supported cooperative work, object oriented programming, shared material, persistent objects.

1 Introduction

In this paper an attempt will be made to bridge two research areas in informatics. These are object oriented programming, or actually object orientation in general, and computer supported cooperative work. This connection is based on the observation that shared material may play an

*Authors addresses: Computer Science Department, Ny Munkegade, 8000 Århus C, Denmark; E-mail paal@daimi.dk or from some countries paal@daimi.uucp; phone +45-6-127188

important role in cooperative work and that object orientation will be useful when implementing computerised shared material. This paper is based on the Mjølner project [5] and on the research programme on computer supported cooperative work [1] at Aarhus University.

The Mjølner project aims at developing an industrial prototype of a programming environment for object oriented programming. The project is deeply rooted in the "Scandinavian approach" to object oriented programming, a fact which also is reflected in the programming languages which the Mjølner environment is going to support: Simula [4] and Beta [12]. In this approach to object oriented programming a program execution, or actually a computer system, is seen as a model of an external system, the referent system: The phenomena in the referent system, the objects, go through their life processes, interacting with each other in various ways. The lives of these objects are reflected by the lives of corresponding objects in the computer system. Thus the ability to write programs where a set of objects execute in parallel is not an extravagant feature of some exotic programming languages; it is one of the most basic means of expression in programming. Correspondingly the semantics of a program is the interpretation of the program as a physical model. More in-depth expositions of this view on object oriented programming can be found elsewhere, see Nygaard [19], Knudsen and Madsen [11], and Madsen and Møller-Pedersen [17].

During the last couple of years computer supported cooperative work has started to receive attention [3], see also the special issues of the *ACM Transactions on Office Information Systems* (5(2), April 1987) and of *Office: Technology and People* (3(2), August 1987).

The term cooperative work is somewhat ill-defined. One tentative definition is given by this author in [24]. Cooperative work is nothing new, but now its existence has been recognised by people working with computers. It turns out that new technical possibilities open up if we may assume that the users are few in number and geographically close. In the transaction cost theory organisations are classified as markets, bureaucracies, and groups or clans [20]. Currently the state of the art of commercial computer systems is that they either support individual work or work divided in a formal manner as in bureaucracies and markets. Support for markets and bureaucracies has typically taken the shape of transaction processing systems or management information systems. The fact that people share tasks, work closely together in teams, know each other, exercise mutual solidarity, etc., is not only unreflected in current computer systems. It is often hampered by the way the computer systems change the conditions of work. The interest in computer supported cooperative work can thus be seen as an attempt to deal with some of the shortcomings of current computer systems.

Computer systems suitable for cooperative work can take many forms. Two main classes of systems are those supporting *explicit communication* and those providing — or supporting the existence of — *shared material.* Typical examples of systems supporting explicit communication are electronic mail, bulletin boards, and synchronous media like the UNIX commands `write` and `talk`. Support for shared material is more subtle, but the support for idea generation and shared writing provided by Cognoter is one example [6,26]. Shared material is the only aspect of computer supported cooperative work dealt with in this paper.

Jackson has written a system development method which is based on object orientation [9]. This method proposes an approach where the basis of the computer system is a model of the referent system. A crucial step in the development process is to identify which kinds of objects the system should model, and which objects are "outside model boundary." The method consists of several steps. After the kinds of objects that are to be modelled have been identified the model is constructed. Thereafter functions may be added to the model. Jackson argues that a model is more stable than the concrete functions, and that a system built in this way will be amendable to new kinds of functions. This requires that the model is sufficiently general, and that the new functions lie within the functionality the model has been made for. The ideas presented in this paper can be seen as an application of some of Jackson's ideas.

The paper proceeds as follows: The next section discusses the role of shared material in cooperative work. It also presents some examples of computer systems providing shared material. Section 3 presents several arguments for modelling material rather than frequent work procedures; these arguments are not restricted to the context of cooperative work. Section 4 presents some ideas for how the design of a computer system could be based on modelling the material used in the work process. Finally, section 5 identifies some of the requirements of the programming languages and run time systems which are incurred by the use of object oriented programming to model shared material.

2 Cooperative work and shared material

Two ways of coordinating cooperative work can be identified. One is by explicit communication about how the work is to be performed, another is less explicit, mediated by the *shared material* used in the work process. A simple example is the way two people carry a table. A part of the coordination may take place as explicit communication, for example in a discussion about how to get the table through a door. When the table is carried, however, the two people can follow each others' actions because the actions get mediated through the shared material. This coordination is not very explicit, it does by no means involve an explicit exchange of messages. Also, it has been learned. There is a big difference between two persons' first attempt at carrying something together, and the way people with experience do it. The learning is both on part of the individual, and on part of the team. It is crucial to this coordination that the actions of the other actor can be immediately observed or felt, so that appropriate corrective or supplementary actions may be taken. The pattern of cooperation is not fixed, it is often defined by the actors. The material and the situation in general make a wide range of patterns of cooperation possible. For people with computer support in mind it should be noted that the mediation of actions taking place through a table is a mediation with a very high bandwidth. Another example is the way a manual file is used as a shared material. A record in such a file can only be in one place at a time. When a document — a record — is gone, it may mean that somebody is already dealing with the problem in question. It may also mean that some other person is using out-of-date information in a potentially dangerous way. This could be

the case with a medical journal. The meaning of an absent record, or some other sign attached to the material, depends on the context, but it is mediated through the material.

It is a central hypothesis in this paper that the properties of the shared material often determine or strongly structure the pattern of cooperation in a work process. If we look at how shared material is used we will get design ideas which are appropriate for computer supported cooperative work. In this way the concept of shared material is used as a design metaphor, a technique proposed by, among others, Madsen [15,16].

When work is computerised the material people work with is normally changed. As a result some properties of the material which were crucial to cooperation may get lost or changed. One example of such a change is that a record is no longer only in one place at a time. Many of the above mentioned "nice" properties of a manual file may therefore disappear. Another example is the transition to computerised text processing. On the surface the typewriter is replaced by a text processor, a "modern typewriter" which has the convenient capacity of storing the text so that corrections can be made without having to retype the whole document. In reality paper is replaced by magnetic storage. Printouts are just snapshots of a document. The document is *in* the computer. This has many consequences, one of these is that the concept of original is more blurred than ever before. There can be many originals since a copy of a file is just as good as the original, and an original can be perceived to be in many places at the same time. People are able to produce new versions of documents more often, and with smaller changes, than before.

Computerisation of material does not, however, have to have negative consequences. There is also a potential for giving the material new properties suitable for coordination. It is the interpretation of this author that the examples presented below implement shared material.

In Colab at Xerox PARC a number of experimental computer systems for cooperative use have been made [26]. Many of these systems are based on the WYSIWIS (What You See Is What I See) metaphor. The users are in other words given the impression of working on a shared surface. Actions of the other users are visible and pieces of material grabbed by somebody else is typically shown to be reserved or removed by being greyed out. For a discussion of the WYSIWIS metaphor see especially Stefik et al. [25].

Kaiser, Kaplan, and Micallef [10] present an experimental programming environment which allows concurrent update of different well defined parts, modules, of the same program. If inconsistencies arise, i.e. if the definition of a module is changed so that its use becomes invalid, the involved programmers will be informed. The technique used in this environment is attribute grammars, like in the Cornell Program Synthesizer [21,28]. The computations are performed in parallel, however, with one process for each module. Information about changed attributes will only in a few cases need to travel from one module, and process, to another. The points where attributes may enter or exit are well defined, they are explicit import and export statements in the programming language. This makes it possible to distinguish between attribute changes that may affect other modules and changes that are certain to be local.

In UNIX and in many other programming environments facilities have been made to support

controlled access to different versions of a file (a program). Two examples running on UNIX are the Source Code Control System (SCCS) [22] and the Revision Control System (RCS) [29]. These two systems perform a multitude of functions. Many of these functions are specific to programming and to a specific way of representing programs (as text files), but they do both implement a material which has the same property as a paper-based document: It can only be in one place at a time. This is implemented through commands like `check-out` and `check-in`. In addition to version numbering of the different modules symbolic naming is supported. This can be used to name configurations and make it easier to retrieve these at later points in time. Different versions of the same module may be updated concurrently. Different branches in a version tree may later be joined or merged, provided the updates of the common ancestor do not textually overlap.

3 Model material, not work procedures

Newman [18] and Suchman and Wynn [27] describe office work as a mixture of problem solving and work according to known procedures. There is not a sharp distinction between these two kinds of work. Newman states [18, p. 55]:

> Existing procedures are used extensively in solving problems. They suggest manageable subgoals and thus simplify the development of problem-solving strategies. In some cases, only a minor modification is needed to a basically adequate procedure; in other cases, an ad hoc solution is constructed from a number of procedural elements.

In many administrative systems, however, the emphasis has been on automating frequent work procedures. This approach can be motivated by its simplicity, and by the fact that much time is spent on these seemingly trivial work procedures. Hence large and immediate savings appear to be obtainable. There are several arguments against this approach:

- The computerisation of work procedures ignores the learning aspects of the supported work. Intimate knowledge of the composition of the procedures is needed in order to perceive the procedure as what it is: a work procedure which sometimes is used "as is", sometimes is combined with other procedures, and often needs slight modification to be applied to a deviant case. This need not cause trouble for users who know the work procedures from before the system was introduced. New users, however, will have little chance for learning this, and will often perceive their work as that of an operator.

- The view of an office as a collection of machines executing procedures is false. The work can be *approximated* by a number of procedures, but this will not cover the work in the office. The strategy of starting with a few, recurrent procedures will turn into a process of implementing an apparently infinite number of procedures having less and less volume. As the procedures get less typical, they will be harder to describe, making the development process more difficult. Also, since the volume is low, the gains from rationalisation, if any, do not outweigh the

development costs. Thus a strategy which start with large potential savings as the aim, may end up as a money sink.

- Work is not static. It is hard and expensive to change tailor-made systems to new or modified procedures. The degree of reuse of the old pieces of software is often low and conversion can cause large difficulties.

- The view of work as the execution of procedures ignores or underemphasises the cooperative aspects of the work. Cooperation which took place in the manual system is reduced to issues of shared access to databases. In other words, only the formalisable, bureaucratic, aspects of cooperation are supported.

The picture given above is not pure speculation. We get confronted with the consequences of the procedure-automating approach in many situations. Some authentic cases:

- A family reserves berths in the night-train from Oslo to Trondheim. The reservation is made for seven persons, but it is explicitly stated that only six are certain to come, the seventh berth should therefore in some way be independent of the other six.

  When they go to the railway station to pick up the tickets, the seventh person has decided not to go. It turns out, however, that one single reservation has been made (probably the only way to get seven berths next to each other). It also turns out that one cannot selectively "dereserve" one berth from a reservation. The clever clerk in the ticket office decides to cancel the seven berths and thereafter reserve six berths. The cancellation works fine, but the new reservation fails because the train is full! (The seven released berths had probably been taken by a waiting list.) Luckily there was another train to Trondheim with free berths that night.[1]

- As a student I used to have a teaching assistantship (TA) and a research assistantship (RA) at Aarhus University. At the beginning of a new term I decided to drop the TA since time was shorter than money. For some obscure reason the university paid me salary as a TA also in the new term. As a consequence more tax was deduced from my RA salary since my deductions had been "spent" on the TA salary. I made the salary department aware of the error and I also stated that I at any time was willing to pay the money I had received in excess of what I would have received only as an RA. This amount was less than what I had received as TA because of progressive tax deductions.

  This was the start of a long series of complicated transactions. It took three months before it all was settled. Although I complained the matter was settled in such a way that I had to pay back all the money I had received as a TA before my tax deduction as an RA was corrected.

- In a Danish bank a typing error led to the erroneous deposit of an enormous amount, say 100 million kroner, on a customer's account. The error was immediately discovered, and the amount was withdrawn only a few instances after it was deposited. So far so good.

[1] Leikny Øgrim, Institute of Informatics, University of Oslo, told me about this case.

Due to the way interest is computed (deposits take place from the next day, withdrawals from the same day) it was computed that the customer had to pay interest for the erroneous amount for one day, approximately 15,000 kroner, roughly equivalent to a months salary. Consequently the bank informs the customer that his account is grossly overdrawn. The customer, of course, complained, and after some dispute the case was settled.

The problem in these cases is not only bad design of the tasks performed by the system, it is the failure to observe the intimate relationship between procedures and problem solving. The alternative proposed here is that instead of automating the procedures one should model, or implement, the *material* used in the work process. Material, or substance, can in a natural way be modelled as objects. Modelling of substance is one of the defining characteristics of object oriented programming. These objects should have value sets corresponding to the different states of the material. There should be operations on the objects corresponding to the *primitive operations* performed on the material. It is in other words proposed that object oriented programming should be applied to implement computerised shared material.

Primitive operations are the most primitive meaningful single operations modifying the material we can identify. Thus `deposit-money-on-an-account` is not such an operation because it involves several primitive operations, like change balance, record information for the computation of interest, etc. To write a single digit in a bank-book is too primitive. It does only have a meaning using a specific kind of technology. Care must be taken in identifying the material and the primitive operations to model. Much inspiration can be obtained from the different ways the corresponding traditional material is being manipulated, but in general a thorough analysis of the work in question, its purposes, etc., is needed. In the terminology of Jackson System Development [9] this corresponds to the entity-action step and some of the activities which have to precede the application of JSD.

There is a conceptual difference between primitive operations and work procedures, not only a difference in level of detail. A work procedure often reflects a normative view of the work, often as seen by management. Procedures are aggregates of simpler operations. Also, procedures are subject to change, and there may be deviations from a procedure in the actual performance of a work task. Primitive operations are stable, they have not changed, and are not expected to change in the future. Also they cannot meaningfully be divided in yet simpler operations.

The distinction between procedures and primitive operations is different from the psychological distinction between actions and operations which has been used by Bødker in [2]. The latter distinction applies to how a person conceives a task. What is a composite action to one person can be an operation to another, typically a professional. The distinction between procedures and primitive operations presented here is not individual. The distinction is given by the system and is in principle common to all users.

The raw model coming out of a development process focussing on material and primitive operations can be used, but it will be very clumsy for many practical purposes. The raw model should be seen as, and designed as, a specialised programming environment with a user-oriented

or profession-oriented programming language. Machine implementation of many entire procedures should be made in this environment, and these programs or functions should be made available to the users for inspection and modification. Users who are inclined to do so can construct their own programs. The programs can, of course, contain calls to other programs as well as to primitive operations. Besides being convenient in programming, it allows some procedures to be seen as simple operations by some users. This implementation of procedures allows a user to reuse, modify, and combine machine implemented procedures in much the same way as can be done with manual procedures. When needed the user may resort to "manual" execution of primitive operations. The new or modified functions can be seen as incremental changes to the existing system. The system will also be tailorable to the needs of the individual users.

Facilities like this are often implemented by mechanisms like accelerators or macros. The possibility to write shell scripts in UNIX is a typical example. This way of doing it works, but is far from ideal because of the poor syntax and semantics of most macro-facilities.

4 Computerised material: an example

This section will discuss how some important aspects of material can be retained or created when the material is computerised. A train seat reservation system is used as example. The example illustrates how many improvements can be made by carefully implementing a new material. Some aspects of cooperation are also illustrated.

Manual seat reservation systems, as the author remembers them, were based on a number of sheets illustrating the cars in the train; see figure 1. The seats could be checked or the part of the

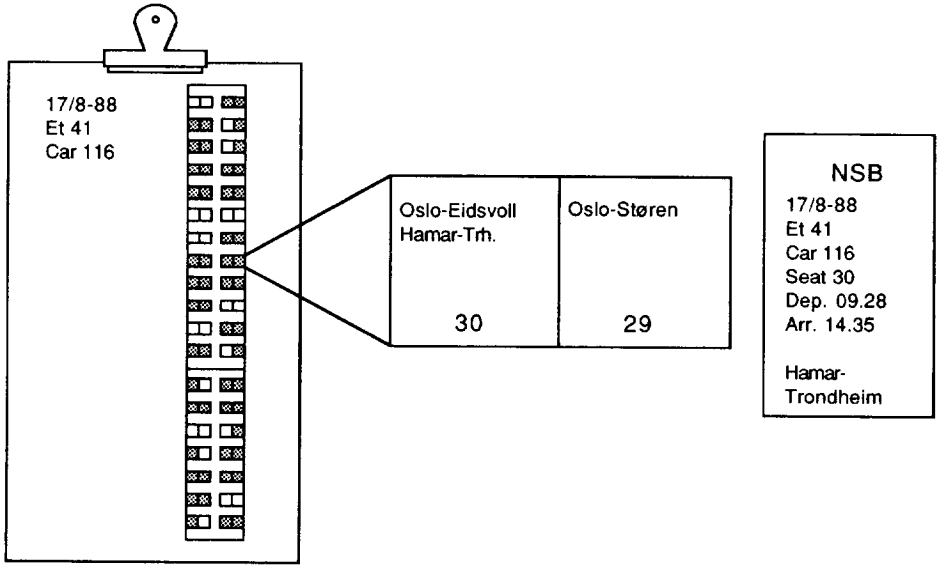

Figure 1: The manual seat reservation system

trip where the seat was occupied could be indicated. Travellers were given small tickets indicating the details of their reservation. This scheme was very flexible in terms of the kinds of services which could be supplied. For example, the reservation of a seat next to another specific seat, and the reservation of a seat close to the door (my grandmother always wanted that), etc., were feasible. Entries were of course written with pencil to allow easy update. The reservation sheets were kept at the train's station of departure so that they could be sent with the conductor.

This system obviously had many drawbacks making it impossible to retain: (The reasons given here are speculation; the author never worked with the railroad.)

- It could not handle high volumes of transactions incurred by compulsory reservation in some trains.
- Updates could only take place at the station of departure. Reservation from other railroad stations and travel agencies had to be made by phone calls to this station.
- The system was inadequate for a new policy where charges were made for seat reservation.
- Reservations from and to abroad were not well integrated in the system.

The system which was implemented solved some of the above-mentioned obvious problems with the manual system. At the same time almost all of the flexibility of the sheet-based system disappeared. It appears that the only functions which were implemented were `make-reservation` and `cancel-reservation`. In fact the only way to figure out whether there was room on a train was to make a reservation.

Many of the necessary improvements could have been made while retaining much of the flexibility of the sheet based system. This could be done by appropriate object oriented modelling of trains with cars and seats, reservations and reservation agents, and by making this model visible to the user. The manual system was actually based on a model of the trains and their seats. A major part of a computerisation of the system should have been to make a computerised model of the trains using the reservation sheets as inspiration for the design.

The approach taken here is different from the approach taken in most database systems. In database systems much effort is made to make it appear as if each user is working alone, unaffected by all others, on the whole database. In the approach taken here the sharing of the material, the reservation sheets, should be made explicit. The users should have access to them in a way which clearly distinguishes between looking at the current state of a piece of information which can be modified by others, and having a unique sheet at disposal for update. Also the process of, in competition with others, obtaining a sheet for unique manipulation must be made explicit. In this way patterns of cooperation can be retained and developed further. In the following a brief sketch will be given of how such a system might look to the user.

Users could have access to shared information by obtaining displays of trains satisfying some criterion, see figure 2. The trains listed in the figure are the trains bringing passengers from Oslo

Trains Oslo S - Hamar 17/8-88

Et 41	0800	0928	43/28/3/2
Ht 351	1010	1157	50/32
Ht 341	1140	1327	25/10
Ht 343	1430	1622	44/35
Ht 375	1542	1733	54/22
Ht 307	1640	1840	11/7
Ht 345	1900	2050	37/22
Ht 405	2300	0049	20/7

Figure 2: List of trains

to Hamar a specific day. The trains are listed with train number, time of departure and arrival, and the number of free seats in different categories. All but one train have only second class. The information displayed here is volatile, and this should be made clear to the users. The best way to do this would be to update the displayed numbers of free seats as reservations and cancellations are made. In this way the users would get an idea of the update activity on the trains, and they could develop their work practices on the basis of that. If a cheaper solution is needed one could display information subject to change in a special font or in a special colour.

Access to "reservation sheets" for single cars can be obtained in a shared display of the train. See figure 3. The figure indicates that car 117 is sold out and that somebody else is updating car

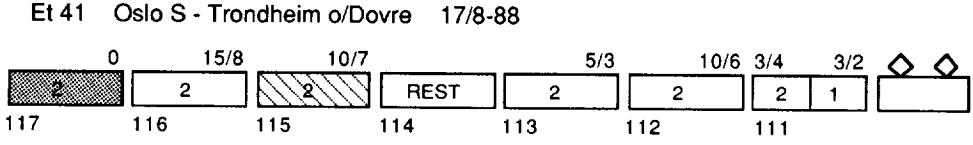

Figure 3: Shared train display

115. The users working with this display compete with each other about obtaining unique access to single reservation sheets. One way to do this would be to implement this display in the WYSIWIS style [25], where the pointing and selection devices of the other users would be visible. In such a setting it could happen that two users try to get the same sheet at the same time. The system does not prevent such collisions, but it must of course detect them. Collision prevention is up to the users, they can see all cursors and can therefore "keep away" from each other. A WYSIWIS solution requires very high bandwidth in the communication network since updates and cursor movements should be propagated in real time between the users. Today such a solution can perhaps only be implemented using a high-speed local area network. This makes it hard to build a seat reservation system that way, but the principle could be applied in a case where the users are geographically close. This would typically be the case in cooperative work. A cheaper solution is to accept that the users do not have any means of seeing each other, and that they therefore will run into collisions more often. The system could also have a built-in function which gets hold of some free car in the train. It will not, however, be acceptable to force users to use this function. It would prevent

services like reserving seats in a car near the restaurant car.

Assume our user successfully gets hold of car 116, see figure 4. The car is now in a private

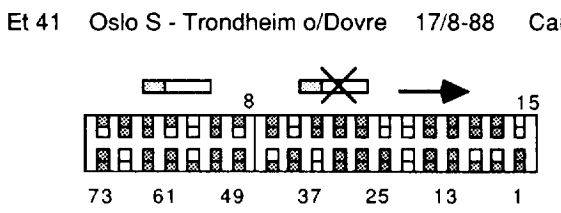

Figure 4: Car display

display, and is displayed to others in the same way as car 115 in figure 3. The user may return the car unmodified to the shared display. This could be the case if the user was searching for a seat near the door and could not find one in this car. The typical mode of operation, however, will be to "enlarge" a part of the car and make a reservation, see figure 5. When the user checks the wanted

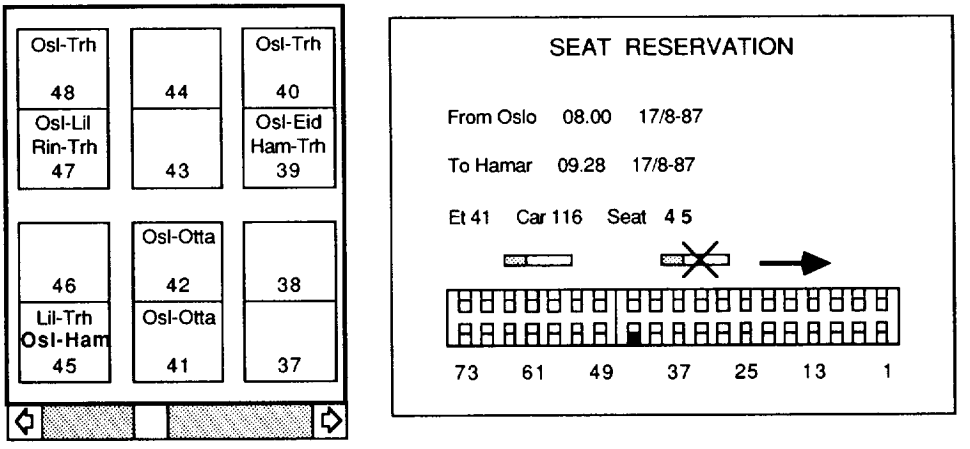

Figure 5: Making a reservation

seats, the corresponding seat reservation is built up in the display. When necessary, seats can be released by updating an old reservation. Standard procedures like `make-reservation` should be available as programmed functions.

It would be too much to say that the example is a system which supports cooperative work. But it does not prevent cooperation either. If cooperation is needed it may evolve because the users to some extent can "see" each other's actions, and because the work organisation is made visible by defining three types of display: a shared information display, a shared selection display, and a private update display.

The grain size of the system, the size of the object allocated to the single user for update, should of course be small enough to allow the needed degree of concurrency. At the same time it should be as large as possible to allow maximal flexibility. In this example the car was made the "grain",

it could also have been the compartment or the whole train. This paper focuses on support for cooperative work, a kind of work where the number of users is limited. A large grain size does not necessarily cause too many collisions in a small group. In addition to this we can interpret collisions as a natural property of the material. It may simply reflect the fact that several users need to coordinate their actions. In computer supported cooperative work the "grain" should therefore be chosen to correspond to a natural piece of material.

Cooperation between users can also be supported by letting two users look at the same reservation, perhaps even work on it concurrently. This can be useful when problems show up or when a case needs to be taken over by another user.

The example shows that the properties of the phenomena which need to be modelled go beyond those which are modelled in traditional computer systems. Especially the "layout" of the train, for example the position of the restaurant car, needs to be reflected in the model. The system needs to keep track of different car types and also their physical layout. This should be done although there is no functionality in the system which depends on this information. Similarly search systems in libraries should contain information about the colour and physical shape of books, although such information do not immediately provide extra functionality.

The example also illustrates that new technology, in this case especially output media which can draw pictures of train cars, result in changes of which parts of the referent system that should be modelled in the computer system. Clearly traditional computer systems have been designed with slow devices with few capabilities and with high communication costs in mind.

It should be asked whether the kind of design proposed here is prohibitively expensive or otherwise unrealistic. Clearly it requires better terminals than those typically found in clerical workplaces, but media with graphic capabilities are getting cheaper. The communication costs involved in transferring drawings of trains, cars, etc., could be high, but they can be reduced by having the different graphics elements distributed once, and later only transfer what is needed to build the drawing in question. In addition, communication costs are also falling.

5 Requirements of the environment

The use of object oriented programming in this paper puts some requirements on the programming languages and the run time environments of the systems.

The objects in question need to be *permanent* or *persistent* across individual program executions. Thus a language with persistent objects is needed, and certainly persistence is more central than object orientation. Reasonable programming languages with no or little support for permanent, in practice disc-based, data structures will therefore have no chance in the competition with 4th generation tools, which can be characterised as data base management systems augmented with primitive programming facilities. We must hope that work on persistent languages and on object oriented databases will change this situation.

Objects used as shared material clearly need to be *sharable* in some well-defined way. In some

cases the sharing will take place as a series of exclusive accesses, in other cases there will be concurrent use of the same objects by several users who make updates and see the effects of their own and others' updates. This issue is discussed by Greif and Sarin in [8] and by Stefik et al. in [26].

In making object oriented models of the referent system *parallelism* arises as a natural part of the model. In a seat reservation system the objects executing in parallel will be the objects modelling the users and, if the system is to be able to handle the reservations while the train is travelling, the objects modelling the trains. Reservation objects can be passive, they only change because some other object sends messages to them. It is therefore important that the language used not only facilitates classes as a kind of abstract datatypes (like Smalltalk [7]), but that the classes have some sort of action part, like in Simula and Beta. Furthermore the objects, i.e. the instances of the classes, must be able to execute in parallel. Parallel execution in Beta is described by Kristensen et al. in [12,14].

Finally these objects may reside at different places in a network of computers. One such place could be an object server. Work is going on to allow the cooperation of several operating system processes to constitute a Beta ensemble [13].

In this paper it has been assumed that it is possible to make a stable model of the shared material. In the seat reservation example it was shown, however, that changes in the model may be needed because new technology makes it relevant to model new aspects of the referent system. Jackson System Development is also based on the stability of the model. New functions are expected to be within the predicted functionality. This will not always be the case. We also know that no description, and hence no model, of reality is complete. If we have a system consisting of many long-lived objects we may therefore need to be able to modify the model while the system is running. In other words: we need *incremental execution*. This may lead to the concurrent presence of instances of different versions of the same class, some mechanisms for handling this is discussed by Skarra and Zdonik in [23]. It is still an open question, however, how we should understand a system which is a changing model of a changing referent system.

In this paper it has been argued that the modelling aspects of object oriented programming can be useful in the implementation of computer based shared material. There is, however, no guarantee that object oriented programming will lead to better computer support for cooperative work. Conversely object oriented programming is not a necessary requirement for making this kind of computer support. The only claim made is that object oriented programming will be *useful* for this purpose.

Acknowledgements

I am indebted to Riitta Hellman and Ole Lehrmann Madsen for their constructive comments.

References

[1] Peter Bøgh Andersen et al. *Research Programme on Computer Support in Cooperative Design and Communication.* IR 70, Computer Science Department, Aarhus University, Århus, 1987.

[2] Susanne Bødker. *Through the Interface — A Human Activity Approach to User Interface Design.* PB 224, Computer Science Department, Aarhus University, Århus, April 1987.

[3] *Conference on Computer Supported Cooperative Work*, MCC Software Technology Program, Austin, Texas, December 1986. Proceedings.

[4] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA 67 Common Base Language.* Pub. S-2, Norwegian Computing Center, Oslo, 1967.

[5] Hans Petter Dahle, Mats Löfgren, Ole Lehrmann Madsen, and Boris Magnusson. The MJØLNER project. In *Software Tools: Improving Applications: Proceedings of the Conference held at Software Tools 87*, pages 81–87, Online Publications, London, 1987.

[6] Gregg Foster and Mark Stefik. Cognoter, theory and practice of a colab-orative tool. In *Proceedings from the Conference on Computer Supported Cooperative Work*, MCC Software Technology Program, Austin, Texas, December 1986.

[7] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, 1983.

[8] Irene Greif and Sunil Sarin. Data sharing in group work. *ACM Transactions on Office Information Systems*, 5(2):187–211, April 1987.

[9] Michael Jackson. *System Development.* Prentice-Hall, Englewood Cliffs, 1983.

[10] Gail E. Kaiser, Simon M. Kaplan, and Josephine Micallef. Multiuser, distributed language-based environments. *IEEE Software*, 4(6):58–67, November 1987.

[11] Jørgen Lindskov Knudsen and Ole Lehrmann Madsen. Teaching object-oriented programming is more than teaching object-oriented programming languages. In Stein Gjessing and Kristen Nygaard, editors, *Proceedings, Second European Conference on Object Oriented Programming (ECOOP'88), Oslo, Norway, August 1988*, Springer Verlag, Heidelberg, 1988.

[12] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48, MIT Press, Cambridge, Massachusetts, 1987.

[13] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Dynamic exchange of BETA systems. January 1985. Unpublished manuscript.

[14] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Multi-sequential execution in the BETA programming language. *Sigplan Notices*, 20(4), April 1985.

[15] Kim Halskov Madsen. Breakthrough by breakdown. In Heinz K. Klein and Kuldeep Kumar, editors, *Proceedings of the IFIP WG8.2 Working Conference on Information Systems Development for Human Progress in Organisation, Atlanta, 29–31 May 1987*, North-Holland, Amsterdam, 1988 (forthcoming). Also available as PB 243, Computer Science Department, Aarhus University, Århus, March 1988.

[16] Kim Halskov Madsen. *Sprogbrug og Design — sammenfattende redegørelse.* PB 245, Computer Science Department, Aarhus University, Århus, November 1987.

[17] Ole Lehrmann Madsen and Birger Møller-Pedersen. What object oriented programming may be — and what it does not have to be. In Stein Gjessing and Kristen Nygaard, editors, *Proceedings, Second European Conference on Object Oriented Programming (ECOOP'88), Oslo, Norway, August 1988*, Springer-Verlag, Heidelberg, 1988.

[18] William M. Newman. *Designing Integrated Systems for the Office Environment.* McGraw-Hill, Singapore, 1986.

[19] Kristen Nygaard. Basic concepts in object oriented programming. *SIGPLAN Notices*, 21(10), October 1986.

[20] William G. Ouchi. Markets, bureaucracies, and clans. *Administrative Science Quaterly*, 25:129–141, March 1980.

[21] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Parctical Software Development Environments*, pages 42–48, May 1984. Published as ACM Software Engineering Notes 9(3) and ACM SIGPLAN Notices 19(5).

[22] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):363–370, December 1975.

[23] Andrea Skarra and Stanley Zdonik. Type evolution in an object-oriented database. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 393–415, MIT Press, Cambridge, Massachusetts, 1987.

[24] Pål Sørgaard. A cooperative work perspective on use and development of computer artifacts. In Pertti Järvinen, editor, *The Report of the 10th IRIS (Information Research seminar In Scandinavia) Seminar*, pages 719–734, University of Tampere, Tampere, 1987. Also available as PB 234, Computer Science Department, Aarhus University, Århus, November 1987.

[25] Mark Stefik, Daniel G. Bobrow, Gregg Foster, Stan Lanning, and Deborah Tatar. WYSIWIS revised: early experiences with multiuser interfaces. *ACM Transactions on Office Information Systems*, 5(2):147–167, April 1987.

[26] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. Beyond the chalkboard: computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, January 1987.

[27] Lucy Suchman and Eleanor Wynn. Procedures and problems in the office. *Office: Technology and People*, 2(2):133–154, January 1984.

[28] Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981. Also in David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors. *Interactive Programming Environments*. McGraw-Hill, New York, 1984.

[29] Walter F. Tichy. RCS: a revision control system. In Pierpaolo Degano and Erik Sandewall, editors, *Integrated Interactive Computing Systems*, pages 345–361, North-Holland, Amsterdam, 1983.