Journal of Econometrics 38 (1988) 227–246. North-Holland

OBJECT-ORIENTED SOFTWARE REPRESENTATIONS FOR STATISTICAL DATA*

R. Wayne OLDFORD

University of Waterloo, Waterloo, Ont. N2L 3G1, Canada

This paper presents new software designs for statistical data. These are implemented using an object-oriented programming paradigm. The implementations are built in a layered fashion from independent representations for the individual, variate, and datum components of a statistical observation to representations for univariate samples and multivariate observations. These in turn are combined according to a software model for an entire data set. This model is quite general and is shown to easily accommodate rather complex data sets. Statistical data-descriptive methods also separate according to the data representations and are therefore incorporated in their definition. How these data representations could be used in a new kind of statistical analysis system is also discussed. In particular, there are some strong implications for interactive statistical graphics.

1. Introduction

In recent years, an important theme has emerged from a variety of subdisciplines of computer science that deal with software engineering. These include such seemingly diverse areas as artificial intelligence, data base design, and programming languages [e.g., see Brodie et al. (1984)]. Simply put, the consensus is that there are strong reasons to recommend building self-contained software abstractions that closely match the natural components of the problem under study. This is particularly important for large software systems that change over time. The point is convincingly made in Abelson and Sussman (1985).

Statistical analysis systems are moderately large software systems. Moreover, it is the nature of statistical data analysis to require extending the system to meet the demands of new problems. It has been argued that this implies that statistical data analysis should be carried out in a highly integrated programming environment [McDonald and Pedersen (1986)]. These are typically found on single user workstations (e.g., lisp workstations) with large real, and much larger virtual, memories where many processes can coexist. This paper presents software representations for statistical data that are designed for such environments. The software described here has been implemented on a Xerox Interlisp-D workstation but could also have been implemented in any other environment supporting object-oriented programming. Elements of this

*Research supported by the Natural Sciences and Engineering Research Council of Canada and by the U.S. National Science Foundation grant no. IST-8420614.

0304-4076/88/$3.50

programming paradigm are discussed as needed in section 3, but see also Abelson and Sussman (1985) and Stefik and Bobrow (1985).

The design of these representations begins with three fundamental, and generally agreed upon, components of statistical data: the variate being measured, the measurement itself, and the individual (or sampling unit) on which the measurement was taken. The components and their properties are described in section 2. In particular, there is a good deal of contextual information wrapped up in what is meant by a statistical observation. As this information can often be crucial to the statistical analysis, it should be recorded on the representation of a statistical observation whenever possible. Section 2 discusses how much of this information can be separated and assigned to each of the three components. Thus circumscribed, each component is given a separate and independent software representation in section 3. To this end, the object-oriented paradigm is used.

More complex data like univariate samples, experimental factors, and multivariate observations are given software representations in section 4 that build on these basic components. These kind of data are traditionally represented as either the columns or rows of some data matrix. However, such representations discard the contextual information available on the individual components. There is additional important contextual information that does not belong to the three basic components, but is naturally associated with the higher level compositions of samples and multivariate observations. And this information is not properly part of a vector's representation. The same is true with many statistical methods. This does not deny the usefulness of vectors and matrices as mathematical constructs, only as accurate representations of statistical data. Section 4 presents new representations of these compound data types. The representations attempt to embody the statistical characteristics that are associated with what they model.

By closely matching the representation to the abstract statistical datatype, the representation becomes a believable token for that statistical abstraction. For example, a single individual could appear in many samples. Consequently, a believable token for this individual would be *single* representation that was accessible from the software representations of all samples containing that individual. This has important implications for quickly accessing up to date information on that individual at different points in the analysis. Since many pieces of software share the representation of the individual, they will always have the most recent information available on that individual. Moreover, if the matching of representation to statistical abstraction is clever enough, this information will be exactly where we expect it.

Just as many univariate samples, or multivariate observations, are combined to produce a data set, so too are their representations combined to give the software representation of a data set. This representation is discussed in section 5. Again, certain kinds of information and methods are associated with data sets and these will need to be incorporated into the representation as well.

It turns out that these representations are enough to represent quite complex data relationships in a natural way. Examples are given in section 6.

How these data representations could be used in future, more integrated, data analysis systems is discussed in section 7. There are some obvious implications for statistical graphics. Finally, some closing remarks are given in the last section.

2. Statistical data

The simplest statistical observation is the value of a variate recorded on some item or individual. As such, it contains a good deal more information than its value alone would indicate.

For example, irrespective of the individual involved, 11.4 centimetres is a suspicious value for the height of a human. This is because the variate (human height), together with the units of measurement (centimetres), gives information that makes the value suspect relative to our prior understanding of the context. Similarly, if '11.4' is instead the gross national product of a nation measured in billions of U.S. dollars, then the nation's identity contains relevant information. For many nations, 11.4 billion U.S. dollars is an unlikely value for the gross national product. Apart from the actual measurement, both the individual on which the measurement is taken and the variate measured provide extra information which can be critical to the statistical analysis.

Indeed it is this kind of contextual information that distinguishes a statistical observation from its familiar mathematical abstraction x_{ij}. For the mathematics, it is enough that the indices distinguish the individuals, i, and the variates, j, but for the statistical analysis, the real world context in which the measurement was taken is crucial. In the transition from the statistical observation to its mathematical representation, such information is discarded.

This contextual information is not regained in the transition from x_{ij} to its software representation as, say, a doubly indexed floating point number. Nor should it be, if the floating point number is to represent a real-valued number. Instead, a statistical observation would be better served by a different software representation, one that incorporates more of its contextual information.

To a remarkable degree, much of the contextual information on a single observation can be cleanly separated into the three sources already mentioned: the individual, the variate, and the measured value. For example, consider a national study where economic variates are recorded for each of many years. Here each year is an 'individual' on which many different economic variates have been measured.

While these measurements will contain most of the information of economic interest for a given individual, there is almost always other information available which is not expressed as the value of a variate. Either the information is not easily expressed as a measurable variate or it is deemed a priori to be of marginal interest to the study. For our example, such information might

be a unique event which occurred in a given year – a war began or ended, a government changed, an international cartel formed, a strike occurred, and so on.

The value of this information is often realized when an examination of the data uncovers a pattern or discrepancy that cannot be accounted for by the measured variates. The extra, non-measured, information is then consulted to explain the pattern or discrepancy. On the basis of such information, some adjustment will likely be made in the subsequent analysis. Perhaps a model will be adjusted, or a new variate introduced (with a value for each individual).

Similarly, each variate has information that is common to the individuals on which it has been measured but which is not to be found in the measurement itself. For example, suppose that the variate is the gross national product. Then a description of what is meant by 'gross national product', how it is to be interpreted, what its relationship is to other variates (a linear combination of some, a possible surrogate for others), and the like, would be important variate information that is independent of any measurement or individual.

Conversely, there is information on the measurement itself that is independent of both the individual and the variate. The value actually recorded, its number of significant digits (if quantitative), whether it is censored (if so, from the left or the right?) are all of this nature.

Since much of the information in a single statistical observation can be separated according to these three sources, it is useful to have a software representation for each. Then, should information be required on a given variate, it can be made accessible from the representation for that variate with minimal, if any, interaction with the representation of any individual or measured value – or anything else. This is especially attractive when different parts of a statistical system access the same piece of information. The software model for a statistical observation is then simply some composition of the three component models.

3. Modelling the components

An object-oriented programming approach is used to model each of the three components of a statistical observation. This approach is based on software constructs called objects.

In brief, each object is a combination of both procedures (called 'methods') and data (stored as the values of 'instance variables'). A class is a structure defined to represent the attributes that are common to many objects. Indeed, every object is defined to be an element or 'instance' of one or more classes. In this way each class gives a template of generic attributes for objects to match. These classes can be organized into an inheritance hierarchy whereby one class 'inherits' all of the procedures and variables of another class simply by naming

the second class as one of its parents. Every class can have multiple parents and multiple children. This turns out to be an extremely effective way to organize the ensuing software – not least because the abstractions the software represents are often naturally organized in this way. The following discussion applies this approach to build object-oriented representations for statistical data. [Stefik and Bobrow (1985) give a more general treatment of object-oriented programming. Oldford and Peters (1988) describe the object-oriented system used here in more detail.]

Using this programming paradigm, the three components of a statistical observation have been modelled as follows. The general concept of each component is represented as a different object class. In particular, the individual or item component of an observation is represented by an object class called **Individual**. Similarly, object classes called **Variate** and **Datum** represent the variate and measurement components, respectively.

The distinction between a class and an instance of a class is now more easily described. Each class is the template for the generic attributes of the component it models. The **Individual** class, then, represents the individual component of a conceptual observation. But an instance of this class represents the individual component of realized observation (e.g., the year 1974, the country France, or the person John Doe). Similarly, an instance of the class **Variate** would be used to represent the variate known as 'gross national product' while the class itself would represent what is meant in general by the term variate. The same can be said for any class and its instances.

The features of a class called 'Instance Variables', or 'IVs' for short, are used to represent attributes that are common to all members of the class but whose value may differ from member to member. For example, the class **Individual** has the IVs 'Label' and 'Notes'. These IVs give slots where individual-specific information that is not expressed as a measure on a variate can be stored. Thus, each instance of **Individual** will have its own name or label as the value of the 'Label' IV, and some a priori description and/or supplemental information learned during the course of the study, stored as the value of its 'Notes' IV.

Procedures, called 'methods', are attached to a class to give its instances relevant procedural behavior. For **Individual**, methods are defined which permit interaction with the values of its IVs (e.g., an 'EditNotes' method).

Like **Individual**, **Variate** has 'Label' and 'Notes' as IVs, and methods, like 'EditNotes', that interact with the IV values for a given instance. These common features are recognized in the software representation by gathering them together to define a new class called **Identifier**. Both **Individual** and **Variate** have all the IVs and methods of **Identifier**, hence each is a special kind of **Identifier**.

This specialization is expressed in object-oriented programming by declaring **Identifier** to be a super, or parent, class of both **Individual** and **Variate**. Fig. 1

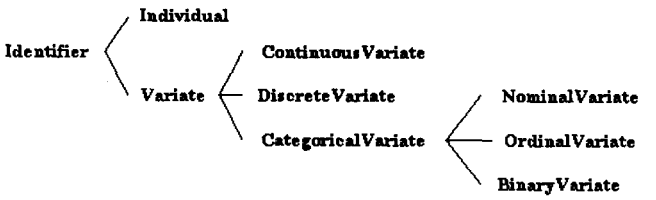

Fig. 1. Inheritance from **Identifier**.

shows the inheritance pattern of all classes specialized from **Identifier** (parent to child = left to right).

Individual and **Variate** inherit all of the properties (IVs and methods) that have been defined for **Identifier**. This means that the IVs 'Label' and 'Notes' need only be specified for **Identifier** – by inheritance they are also IVs of **Individual** and **Variate**.

Variate, however, has IVs like the natural 'Range' of the measurements taken on that variate, which distinguish it from **Identifier**. Moreover, there are distinct kinds of variates that are naturally expressed as specializations of **Variate**.

The first specialization of **Variate** is into three distinct kinds of variates found in practice, namely **ContinuousVariate**, **DiscreteVariate**, and **CategoricalVariate**. Measured values on either of the first two must be numbers, reals for the first and integers for the second. On the third the values must be categories. A **CategoricalVariate** differs from the others in that the value of its 'Range' IV is a list of possible measurement values [e.g., (Favour, Indifferent, Oppose) or (1,2,3,4)].

CategoricalVariate is further specialized into **NominalVariate** and **OrdinalVariate**, depending on whether the order of the 'Range' list is informative. **BinaryVariate** is given special status because of its frequent and important occurrences in practice.

Finally, a **Datum** is meant to contain all the information related to the measurement but none on the variate or individual being measured. The separation is not always clear. Certain attributes like the 'DataValue' (a number or a string), the 'Censoring' (NIL, Left, or Right), and the 'SignificantDigits' (NIL or a positive integer) belong to the value recorded as the measurement and hence are IVs of **Datum**. For others it is not as clear where the attribute belongs. Units of measurement, for example, are part of the recorded measurement, but they are typically the same for all measurements taken on a variate (cf. 'SignificantDigits'). Hence, in the current representation 'UnitsOfMeasurement' and 'Range' are attached to **ContinuousVariate** and **DiscreteVariate**, rather than to **Datum**.

A simple statistical observation will be some union of an **Individual**, a **Variate**, and a **Datum**, where all of the available information is recorded on

each one as appropriate. Note that the same information can be shared by many observations. The same instance of **Individual** can be part of many different observations and hence make the same individual-specific information available to each observation. For example, suppose that, while examining the measurements on one variate, something is learned about a given individual and recorded as 'Notes' on the corresponding instance of **Individual**. This information will then be available when examining a measurement on any other variate for the same individual. Alternatively, a single instance of a **ContinuousVariate**, will contain information about that particular variate that will be available to all observations that are measurements of it. This is an important consequence of modelling the components separately.

4. Putting the pieces together

In this section, software representations are presented for univariate samples, or batches, and multivariate observations. These are more complex statistical data abstractions that are typically represented as either a column or a row of a 'data matrix'. However, if the representations are intended to be believable tokens for the statistical abstractions, this simplistic representation will not do. For example, unmeasured contextual information recorded on a given individual in the sample is not available from the matrix. A believable software representation of a univariate sample must have immediate access to the information available on the measured variate and on every individual in the sample. A representation which first passes the statistical abstraction through the mathematical filter of a vector loses this information. Software representations of matrices and vectors are constructed to be believable representations of mathematical, not statistical, abstractions. A univariate sample would be better represented more directly – as some composition of the basic elements of a statistical observation as described in section 3. The representations of this section follow this approach.

First, features that are common to both multivariate observations and univariate samples can be extracted by noticing that in either case, the statistical data are examined in a conditional fashion. A multivariate observation holds a given individual fixed to consider the measurement of each of a number of variates. Alternatively, a univariate sample holds the variate fixed to consider the measurement on each of a number of individuals. An object class called a **DataRecord** is defined to represent the common features of the statistical abstractions of a multivariate observation and a univariates sample. **DataRecord** is based on the conditional nature of these statistical data abstractions. In either case, a single kind of **Identifier** is held fixed (**Individual** or **Variate**) and a collection of **Identifier–Datum** pairs (**Variate–Datum** or **Individual–Datum**) examined.

A **DataRecord** pairs an instance of one of the two **Identifier** types (an **Individual** or a kind of **Variate**) with a collection of paired instances of an

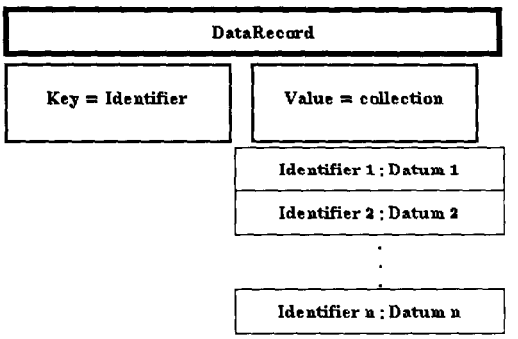

Fig. 2. A **DataRecord**.

Identifier of the other type (a **Variate** or an **Individual**) and a **Datum**. A schematic description is given in fig. 2.

As fig. 2 indicates a **DataRecord** has two distinct parts. There is the **Identifier** (either an **Individual** or a **Variate**) which is fixed and is called the 'key' of the **DataRecord**. And there is the collection of **Identifier–Datum** pairs that is called its 'value'. In fig. 2 there are n such pairs. The kind of **Identifier** in each of these pairs is of course different from the **Identifier** that is the **DataRecord**'s key. Further, each **Identifier** in the collection is unique.

Many actions are naturally associated with a **DataRecord**. For example, we expect to be able to identify the key **Identifier** and to access and put information on it. This is simply accomplished by a method on the **DataRecord** class which permits access to the key and hence to its methods. Similarly, methods are attached to **DataRecord** to interact with its 'value'.

In particular, many methods are expected to be attached to **DataRecord** because it is a keyed-collection of pairs. Those described in Goldberg and Robson (1983) for a keyed-collection have been implemented in Interlisp-D for the class **DataRecord**. These include interactions such as adding and removing pairs, accessing one element of a pair by specifying its partner, iterating over the collection to apply a function to each pair, and many others.

Other interactions are appropriate because a **DataRecord** represents an indexed collection of statistical data. Methods like 'SelectCensored' (which returns another instance of **DataRecord** containing all data pairs whose **Datum**'s 'Censoring' is non-NIL) are therefore included in the definition of **DataRecord**.

These methods treat a **DataRecord** only as a convenient means for interacting with each **Identifier–Datum** pair of its collection. None, so far, treats a **DataRecord** as the unit. But this is contrary to the abstraction a **DataRecord** represents – a multivariate observation or univariate sample. For example,

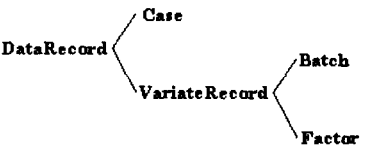

Fig. 3. Specializing **DataRecord**.

prior theoretical information may indicate that a particular batch of numbers should appear as a sample from a specified distributional family. This information quite properly belongs to the **DataRecord** representing the batch rather than to, say, the **Variate** which is its key. The IV 'Notes' and the method 'EditNotes' are therefore attached to **DataRecord** to allow information on the **DataRecord** as a unit to be recorded.

Note, however, that many methods which one might naturally associate with certain **DataRecords** are not appropriate for all **DataRecords**. For example, when an instance of **DataRecord** is used to represent a batch of numbers for a single variate, we expect to be able to display that data as a boxplot. But for another instance that represents a single multivariate observation, a boxplot makes little sense. The two represent substantively different kinds of **DataRecords**.

To make the representations truer to the statistical abstractions they model, the **DataRecord** class is specialized as shown in fig. 3. The two principal types of **DataRecord** are the **Case** and the **VariateRecord**. These correspond directly to the two substantively different statistical abstractions of a multivariate observation and a univariate sample respectively. A **Case** is a **DataRecord** whose key is always an **Individual** and whose value is always a collection of **Variate–Datum** pairs. Likewise, a **VariateRecord** is a **DataRecord** whose key must be a **Variate** and whose value must be a collection of **Individual–Datum** pairs.

Similar considerations lead to a subdivision of **VariateRecord** into **Batch** and **Factor**, according to the kind of **Variate** taken as the key. A **Batch** must have either a **ContinuousVariate** or a **DiscreteVariate** as its key, whereas a **Factor** must have a **CategoricalVariate**.

This division of **VariateRecord** is a natural one to which many statistical methods adhere. Methods which produce numerical summaries like the mean, the standard deviation, empirical quantiles, and so on, are applicable to a batch of numbers. Hence they are part of the definition of **Batch**. The same is true for re-expressions, such as the natural logarithm, which are applied to all **Individual–Datum** pairs of a **Batch** to produce a new **Batch** having the same **Individuals** but different **Datums**.

By contrast, re-expressions make no sense for a **Factor**. Numerical summaries are restricted to such quantities as the mode, the counts for every category,

and, if the **CategoricalVariate** is an **OrdinalVariate**, perhaps some quantiles as well. Consequently, these methods are associated with the class **Factor**.

Statistical graphics also separate according to the inheritance structure from **DataRecord**. A **Batch** has methods which will display it as a boxplot or as a histogram. But a **Factor** will be displayed as a barplot. Quite different displays make sense for a **Case** – from a circle in a scatterplot, to a glyph, to a cartoon face.

Each kind of **DataRecord** – **Case**, **Batch**, and **Factor** – represents a unique statistical data abstraction and the associated methods separate accordingly. Each **DataRecord** is defined as a unit with which certain kinds of information and interactions belong. However, when detailed information is required for the components of a **DataRecord** – **Individuals**, **Variates**, and **Datum** – the components are accessed or interacted with directly.

Moreover, some instances of these components will be shared by many different **DataRecords**. Information stored on an **Individual** after examining some **Case** is accessible from any **Batch** which contains a measurement for that **Individual**. The design of **DataRecord** as a composition of more primitive objects makes this possible. The same principle is applied in the next section to create a representation of a data set as the composition of instances of **DataRecords**.

5. DataSets

A typical statistical problem begins with many batches, factors and cases that can share variates, individuals and data values. Collectively, they constitute the base data set. Similarly, their software representations as **Batches**, **Factors**, and **Cases** can be gathered together within an instance of an object class called **DataSet**. **DataSet** is designed to represent the higher level statistical abstraction of a data set. An instance of **DataSet** would, for example, be one of the basic inputs to a regression analysis (along with a fitting procedure and a model representation involving **Variates** and parameter representations).

Like **DataRecord**, **DataSet** is a collection of data. A **DataSet** collects together many instances of **Case**, indexing each one within the **DataSet** by the **Individual** that is its key. Fig. 4 illustrates the organization. Just as a **DataRecord** is a collection of **Identifier–Datum** pairs, a **DataSet** is a collection of **Individual–Case** pairs. Indeed, this commonality means that the implementations of **DataRecord** and **DataSet** are very similar in structure. [Oldford and Peters (1986) describe the implementation in more detail.] Note also that unlike using a rectangular array to represent the data, there is no need to record missing values explicitly as NAs – they are simply indicated by their absence.

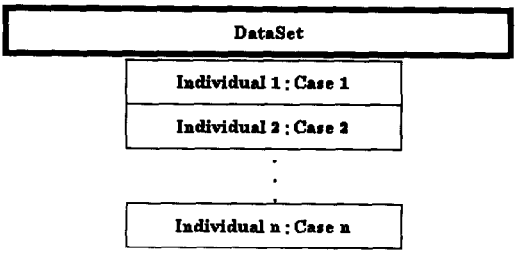

Fig. 4. A **DataSet**.

A **DataSet** can access each of its **Cases** (their IVs and methods), and, through them, information on the **Individuals**, **Variates**, and **Datums** they contain. Like **DataRecord**, there will also be information that more properly belongs to a given **DataSet** than to any of its components (e.g., the source of the data, how the data was gathered, a description of the goals of the study, etc.). Hence IVs like 'Label' and 'Notes', and their associated methods, are also defined for **DataSet**.

Again, as with **DataRecords**, the functions that one expects to perform on any collection of keyed-objects are available as methods for a **DataSet.** These include iteration over the entire collection to select certain elements, or to test each element for a given feature, or to apply some user-specified function to every element (the elements referred to here can be either the **Individuals** or the **Cases**). The complete methods of Goldberg and Robson (1983) are installed.

However, **Cases** are not always the most natural unit with which to work. Often data are not gathered, or considered, **Case** by **Case**, but rather **VariateRecord** by **VariateRecord**. For example, economic data are often gathered series by series from different data sources. In practice, then, data sets are viewed symmetrically as either a collection of **Cases** or as a collection of **VariateRecords**, depending on which seems most helpful for understanding the problem at hand. A **DataSet** is therefore defined to enjoy the same property.

The contents of whole **VariateRecords** may be added to, or removed from, a **DataSet** as easily as are **Cases**. Similarly, iteration, testing, and so on, can be done over **Variates** as easily as over **Individuals**. Furthermore, wherever possible, the same method is used to interact with a **DataSet** either as a collection of **Cases** or as a collection of **VariateRecords**. For example, the method 'AtPut' allows information 'At' a specified location in the **DataSet** to be accessed. It returns a **VariateRecord** if the argument supplied is a **Variate**, a **Case** if the argument is an **Individual**, and a **Datum** if both a **Variate** and an **Individual** are supplied together.

This symmetry is not, however, completely enforced in the present implementation of **DataSet**. It is not possible, for example, to identify whether a

DataSet contains a particular instance of a **Batch**. This is because a **DataSet** does not contain instances of any kind of **VariateRecord**. It is implemented only as a collection of **Case**s. Therefore, the most that can be determined is whether the **DataSet** contains the same data as does the **Batch**.

It could easily be otherwise, but this would require a double referencing scheme in which **Case**s and **VariateRecord**s are used simultaneously in the definition of a **DataSet**. **Case**s and **VariateRecord**s would need to be automatically updated whenever the contents of one or the other changes. This could be done by employing active constraints between the **Case**s, **VariateRecord**s, and **DataSet** [e.g., see Borning (1981) or McDonald (1986)]. The next implementation will probably have this double referencing scheme.

Aside from the usual methods for any new kind of collection, there are also statistical methods that are associated with an entire **DataSet**. All pairwise plots of the data for selected **Variate**s, a plot of the Chernoff faces for selected **Individual**s and **Variate**s, boxplots of the data for selected **Variate**s, various measures of multivariate location and dispersion, are all natural candidates for methods of **DataSet**. The key common feature is that each method produces only descriptive information on the data.

Other methods, like one to perform a multivariate regression of selected variates on other selected variates, are not attached to **DataSet**. This is because its appropriate application depends on information that is extraneous to the recorded data. This includes information such as the purpose for fitting such a model, the structural relationship between the variates, and the probability model, if any, that can be assumed to have generated the data. Lagged variates, for example, do not appear in the representation of the data. Lags are not part of the data representation but rather part of the representation of a model relating one or more variates. A model specifying this relationship, together with the **DataSet** for which the model is believed to hold, are the basic inputs for estimation. **DataSet**s, like other representations of statistical data, are meant to model only the data.

It is debatable whether statistical methods should be attached to these representations at all. If so, then where the boundary of the data lies is not always easily determined. For example, should density estimates be attached to a **Batch**? The guiding principle here, is that, only if its purpose is purely descriptive should the method be attached to the data. For a **Batch**, it is enough that a 'reasonable' density estimate be available. One that is optimal, in some sense, is not really of interest – indeed, a good interactive histogram will probably suffice. If, however, the purpose is to construct a reliable density estimate for some theoretical probability distribution that generated the data, then the problem is genuinely one of estimation (or modelling) and is more properly addressed by statistical procedures which use the **Batch** as input. Those statistical methods which are directly attached to the data representations should not require information that is extraneous to the recorded data.

6. Modelling more complex data relationships

So far, the data information considered has separated quite neatly into the three components: individual, variate, and datum. From these, increasingly abstract datatypes have been built up, layer by layer, from the three base components to **DataRecords** (**VariateRecords**, **Batches**, **Factors**, and **Cases**) and from **DataRecords** to **DataSets**. In this section, two examples of more complex data relationships are considered to see how they would be represented using these datatypes. Some extensions to the datatypes are also suggested.

The simplest example that does not obviously fit the data representations proposed in previous sections is time-ordered data. To give a concrete example, consider how the US gross national product recorded for the years 1947 to 1974 would be represented.

To start, each year is represented as an instance of **Individual** on which an instance of **ContinuousVariate** called 'US gross national product' has been measured. For each **Individual**, the measurement is stored in a corresponding instance of **Datum**. However, if these data were now collected together as an instance of **Batch**, then the ordering of the **Individuals** would be lost! **Batches** do not record order.

To avoid any loss of information, another instance of **Variate** (**ContinuousVariate** or **DiscreteVariate** would do) is introduced to represent the time. Then a new instance of **Datum** is introduced for each **Individual** and its value would be the year (1947,1948,...,1974) that corresponds to that **Individual**. The series would now be represented as an instance of **DataSet**, where each **Case** in the **DataSet** is composed of the **Datums** associated with two variates: the gross national product and the time.

Note that since the year is recorded as the value of a **Variate**, the instances of **Individual** have been reduced to symbolic place holders. Hence, any potentially important events in a given year would be recorded on the **Datum** that corresponds to that year and not on the **Individual**.

An alternative way to represent a time series – one that is especially appealing if one works a great deal with time series – is to introduce new classes to represent time-ordered data. Simply specialize **VariateRecord** to a new class of **VariateRecord** called **TimeSeries**, say, whose **Individuals** are ordered within the **TimeSeries**. Similarly, **DataSet** could be specialized to **MultipleTimeSeries**. The effect would be the same as representing the time series as a **DataSet** with a time **Variate**, but with an important difference: since a new class is defined, new methods specific to that class (e.g., 'ProduceAPeriodogram') can be attached to it.

Unfortunately, in either representation, each **Individual** no longer represents the unit sample – even though the time is usually regarded as just that. A third approach would be to incorporate more information into the definition of the class **Individual** (or specialize **Individual**) to give its instances the ability to

order themselves as necessary. Then if the year is regarded as the sampling unit on which the **Variate** is measured, the instances of **Individual** representing each year would have some method to establish order. This could easily be accomplished by adding two new IVs, say 'CompareFunction' and 'ComparisonValue', to the **Individual** class. If two instances have the identical value of 'CompareFunction', then it could be applied to the values of their 'ComparisonValue' to arrive at an ordering. For 'gross national product' time series, the comparison function would simply be 'greater-than-or-equal-to', and the comparison values would be the numbers 1947,...,1974.

The appeal of this approach is that it does not confuse the sampling units (the individual times) with a hypothetical variate. Rather it is based on time-ordering being a property of the sampling units (the times) and hence a part of the definition of **Individual**. Moreover, it does not preclude using either of the previous representations for time series.

A quite different example is given by the situation where there are repeated measurements on the same individual and many individuals under study. Again to be concrete, suppose that the amount of carbon dioxide dissolved in a sample of blood is the variate measured. Then suppose that this measurement is taken from each of many independent samples from the same person and that there are many such persons in the study. How should the data be recorded?

One approach is the following. First, represent each blood sample by an instance of **Individual** and the **Variate** 'amount of CO_2 dissolved' by an instance of **ContinuousVariate**. Then, use an instance of **Datum** to record the measurement of this **ContinuousVariate** for each blood sample. Similarly, represent each person in the study by an instance of **Individual**.

To group the blood samples by the donor, an instance of **NominalVariate** is used. The instance of **Datum** associated with this **NominalVariate** would then have, as its value, that instance of **Individual** that represents the donor. Moreover, all measurements on the same person could share the same instance of **Datum** for the **NominalVariate** (since it has the same value). The representation for the whole data set is now given by a **DataSet**, where each **Case** has as its key an **Individual** corresponding to a blood sample. The collection for each **Case** contains the **Datums** on the **ContinuousVariate** and **NominalVariate** for that blood sample.

The appeal of this approach is that each sampling unit (both persons and blood samples) is represented quite naturally as an instance of **Individual**. However, the use of a **NominalVariate** to nest one sampling unit (the blood sample) within the other (the donor) seems artificial and forced.

A preferable representation of the data that does not use a **NominalVariate** in this way is the following. As before, represent the blood samples and donors as separate instances of **Individual**. Also, retain the **ContinuousVariate** to represent the 'amount of CO_2 dissolved' and all the **Datums** used to record the measured values. Again the data will be represented by a **DataSet** of **Cases**.

But this time, a **Case** is constructed for those instances of **Individual** that correspond to a donor, not a blood sample.

Each **Case** contains only one **Variate–Datum** pair. The **Variate** is the **ContinuousVariate** just mentioned, but the **Datum** is quite different. This **Datum** has an instance of **Batch** as its value. The **Batch** corresponds to the collection of blood samples, and the amount of dissolved CO_2 measured for each, that were taken from the donor represented by the **Individual** for the **Case**. That is, the **Batch** has as its key the **ContinuousVariate** and as its value the collection of **Individual–Datum** pairs that are appropriate.

This representation has the same appealing attributes as does the previous one. But unlike the previous one, no artificial variate was introduced. More importantly, the **Individuals** in this representation are nested within the **DataSet** in a manner that is completely analogous to the nesting of the sampling units they represent. It would seem, therefore, to be the preferable representation of the two.

Both representations for the repeated measures data can be constructed using the framework that has been implemented. While the second style is preferred (on the grounds that it more closely follows the way the data are usually recorded), nothing in the implementation prevents one from using either. Moreover, both styles of representation generalize in a straightforward manner to model more complex data relationships reasonably.

However, in either style of representation, no information is attached to an **Individual** that directly related it to any other **Individual**. That is, no information is directly accessible to an **Individual** that could show how it is related to any other. Neither **Individuals** nor **Variates** have a link to any other object. The relationships between them are available only at a level that encapsulates more than one of them (e.g., **DataRecord**). Consequently, in the above representations, this information is only available from the instance of **Case** that connects them.

To recognize explicitly that sampling units can be related to one another requires a richer definition of **Individual** than that considered here. How such structures are best defined is an open problem. If solved, though, it could mean quick access to relevant sampling unit information from any **Individual** in the study. Similarly, rich structures for **Variate** could allow the variate relationships to be tracked in the analysis (e.g., one is a linear combination of others, or is a transformation of another, and so on).

For more complex data scenarios, **DataSets** could be redefined to more closely match the **DataRecord** pattern by including a 'key' for each **DataSet** (compare figs. 2 and 4) and then allowing a collection of **DataSets** as its contents. The value of the key would be an **Individual**. An example, where this could make sense is the situation that arises when many variates are measured on many persons over time. There would be a single **Variate** for each variate measured, a single **Individual** for each person, and a single **Individual** for each time (the **Individuals** that correspond to time would have the order information

attached to them). Such a data set could be represented in two equivalent forms with extended **DataSet**s. The first would have a **DataSet** for the collection of measurements on each person measured at a given time. A key that is the **Individual** representing that time would be attached to this **DataSet** and the collection of **DataSet**s would be gathered together in a larger extended **DataSet** keyed by the time. The alternative representation would simply switch the roles of the **Individual**s representing the times and those representing the persons.

7. Some implications for data analysis systems

An important feature of these data representations is that they share information. When all the instances reside in the virtual memory of the machine (at least for the duration of the analysis) the updating of information on any one of them is immediate. Moreover, the new information becomes immediately available to all software (including other instances) that point to the updated instance.

As an example, suppose that in the course of the analysis of some US annual economic data, it is observed that some significant change in prices happened in 1974. Upon reflection, it occurs to the investigator that this is probably attributable to the fixing of world oil prices by OPEC in that year. This significant event is therefore recorded on the 'Notes' IV of the **Individual** that represents the year of 1974.

Later, another economic analysis is underway. A different country is being studied, or different variates on the same country, or perhaps only the investigator has changed. In any case, suppose that the same years are involved. Since the **Individual**s of the previous study represent these years, the same **Individual**s should be used for this study. Then, if the data associated with 1974 stand out in the second study too, the information about OPEC previously stored on that **Individual** will be available to the investigator for consideration. It may now be accessed through a different instance of **DataSet**, but it will be precisely the same **Individual** as before and hence the same information.

This has a subtle, but important, effect on one's expectations of the statistical analysis system. For example, we no longer expect to retrieve information on an **Individual** that represents a person from two different **Batch**es simply because we know it is possible to move information from one **Batch** to another. Rather, we expect to be able to retrieve that information because we are referring to a single person! The **Individual** becomes more closely identified with the person it represents. Each component part, at every level (from **Datum** to **DataRecord** to **DataSet**), is designed to model a unit that is natural in statistical data analysis. Having the same software representation appear in those data structures where the corresponding natural unit of information could be expected to appear, strongly reinforces the identification

of the unit with its software representation. The expectation, then, is that, wherever the statistical unit appears naturally in the course of the analysis, its unique software representation should be available.

Consider, for example, how this expectation makes new demands on the organization of interactive statistical graphics. It was noted earlier that many statistical graphics separate along the same lines as the representations for statistical data. Boxplots and histograms are appropriate for **Batch**es, barplots for **Factor**s, glyphs and Chernoff faces for **Case**s, and pairwise scatterplots and side by side boxplots for **DataSet**s. And so, methods to produce many of these plots are part of the definition of each representation. It is no accident that the graphics neatly match the data units; they were originally developed with those units in mind. When we look at a boxplot, we are looking at the display of a batch of numbers. The expectation, then, is that from that display we should be able to access all information on that batch. To do this, the software that represents the boxplot display should have access to the **Batch** from which it was built.

The kind of direct connection that is expected becomes obvious when a simple scatterplot of N observations on two variates is considered. The data are represented as an instance of **DataSet** that has N **Case**s, say **Case**-1,**Case**-2,...,**Case**-N. In general, **Case**-i has as its key an instance of **Individual**, say **Individual**-i, and as its value a collection of at least two **Variate–Datum** pairs. Let these two **Variate**s be denoted **Variate**-A and **Variate**-B. Then there will also be at least two **Batch**es associated with this **DataSet**, **Batch**-A and **Batch**-B, representing the N **Datum**s for **Variate**-A and **Variate**-B, respectively.

The whole scatterplot is a display of this **DataSet**. From this display we expect to be able to access all the available information on the **DataSet**. In a highly interactive environment with a high-resolution display and a pointing device like a mouse, this information might be accessed by simply pointing at the scatterplot with the mouse, pressing a button, and selecting menu items to choose the pieces of information desired.

However, interest in a scatterplot often lies in the individual observations. The ith point is essentially a view on **Case**-i. Hence, direct interaction with that point in the display is a natural way to obtain information on either **Case**-i or **Individual**-i (possibly positioning the mouse over the displayed point and clicking a button to produce a menu giving access to **Case**-i and **Individual**-i). Conversely, since information is also gained on **Case**-i by examining the scatterplot, interacting with the displayed point would be a natural way to store information directly onto **Case**-i or **Individual**-i.

Moreover, if each point really is a view of a **Case**, and there are many possible ways to view a **Case**, then why not choose the most appropriate display interactively. For example, choose different symbols (cross, circle, triangle) for each **Case** on the basis of the value of a **NominalVariate** taken in the **Case**. Or display each **Case** as a glyph to represent the values on the remaining **Variate**s in the **Case**.

Other elements of a scatterplot also correspond to views of data structures. Axes are views of the marginal batches – interacting with them should lead to interacting with **Batch-*A*** and **Variate-*A*** or with **Batch-*B*** and **Variate-*B***. Again, other views of these **Batches**, like boxplots, range-lines and histograms, should be exchangeable in the scatterplot.

A scatterplot, then, is just an arrangement of selected views of the internal data structures of a **DataSet**. As such, it is natural to expect to manipulate the data structure and change how it is displayed by interacting directly with the display. Moreover, we would expect all the views to be current. That is, if changes are made to an underlying data structure (e.g., change in data values, addition of new variates and values), then every view of that data structure is expected to update its display to reflect the change. This includes the situation where one data structure is simultaneously displayed in different plots (e.g., all pairwise scatterplots for a **DataSet**, each **Case** is displayed many times). Changes to the underlying data structures should be immediately reflected in all plots that display it. Effectively, then, each view is a link from the plot to the data structure it displays.

Moreover, if each of these views is also given an object-oriented representation (**CaseView**, **BatchView**, etc.), then they too could be shared by two or more different scatterplots. Suppose, for example, that **CaseView**-1 is the view of **Case**-1 and that it is displayed as a cross in two different scatterplots. Changing its display to a circle in one plot should cause it to change in the other plot as well. In this way, the plots themselves are linked by sharing the same **CaseView**-1 [see also Becker and Cleveland (1987)].

In a system called Antelope, McDonald (1986) shows that simple leader–follower constraints are enough to achieve this kind of linking between scatterplots. Basically, one object called the follower updates its values whenever another object called the leader changes its values. In the above situation, two levels of constraints need to be applied. For the first, the leaders are taken to be **Case**s and the followers **CaseView**s. This ensures that whenever the basic data in a **Case** changes, its **CaseView** is updated. The second level then constrains the objects representing the scatterplots to follow the **CaseView**s. Then when **CaseView**-1 changes, the two scatterplots update their displays.

While all statistical graphics display views of data structures, some often display other things as well. Items like a running-linear-smooth of the data, or a fitted curve are sometimes added to simple scatterplots. Everything in the display, though, is a display, or view, of something. The displayed curve might be a view of the result from fitting a parametric model to the displayed data by least squares. The smooth might be a view of the results from a local least-squares fitting of a straight line. If there is a corresponding software representation for items being viewed, then there is reason to interact with them as well.

This illustrates an important point hinted at earlier. Many statistical structures which are not data structures, can be usefully represented in an object-oriented fashion. For example, representations of statistical models as data structures have recently been considered by Bates and Chambers (1988). In the present framework, if the mathematical specification of a model describes the relationship between different variates, then its software representation should relate specific instances of **Variate** classes. Since the mathematical specification has nothing to do with the observed data, then its representation can exist independently and be fitted to many different **DataSets**.

In the DINDE system, many different kinds of statistical structures are given object oriented representations [Oldford and Peters (1987)]. These include familiar graphics like **Scatterplots**, **QQPlots**, **ResidualPlots**, **Histograms**, and **Boxplots**, and less familiar analysis artifacts like **LinearFits** and **LeastSquaresFits**. As was the case with the data structure, the key to these representations is to isolate a conceptual statistical structure (e.g., a regression model specification or the raw information gained by fitting a model to data with least-squares) and to represent it as an object class (e.g., **LinearRegressionModel** or **LeastSquaresFit**). If the representation is good, instances of it will be used in the analysis as believable tokens for that statistical information.

As with **DataSets**, these richer statistical objects, if properly defined, will be able to share a great deal of information with one another (and with **DataSets**). A representation for a mathematical model, say **Model**, would need access to the **Variate** instances for which it describes a relationship. A **LeastSquaresFit** would need access to the **Model** used to get the fit and to the **DataSet** on which the fit was based. If the same mathematical relationship is fitted using different data, then a different **DataSet** is used and a different **LeastSquaresFit** results. But the same instance of **Model** would be shared by both fits, and hence the **Model** specified instances of **Variates** must be present in both **DataSets**.

If the selected concepts are natural, well-defined units in statistical data analysis, and they are believably represented by the software, then there will be increased pressure on integrating the statistical software. As before, the expectation will be that wherever the statistical unit appears naturally, so too should its unique representation.

8. Concluding remarks

The data structures that have been represented here are quite general. They focus on attributes of statistical data that are common to most statistical problems. While more specific data structures, like experimental runs or time series, can be modelled using these representations, they are often regarded as containing information beyond the usual notion of multivariate observations or batches. The given data classes have been designed to be generic enough

that specializations of them could be used to give a suitable representation of more specific data types (e.g., **TimeSeries** as a specialization of **VariateRecord**).

Other data structures like repeated measures data, while representable within the present framework, suggest that the basic components themselves might be improved upon. Some statistical data information is on the organization of the sampling units, independent of any variates or measurements. Some units are nested within another, or perhaps crossed with others. This organization might be bested modelled separately, perhaps by extending the definition of **Individual** to admit relationships between different instances of **Individual**.

A statistical data structure that is represented as a unique instance of some class can become strongly identified with that instance. But it requires that that instance be shared by different programs. For example, the same instance of an **Individual** will be found within different instances of **Batches**, **Cases**, and others. Once this identification occurs it is readily extended to interactive statistical graphics as well.

Finally, with important elements of statistical data and graphics modelled well, other units of statistical analysis that are based on them could be considered. Units like models, fitted relationships, and specialized plots, which are used frequently in statistical data analysis, have enough properties that are invariant from one analysis to the next, that their software representation is conceivable. Again integration with the existing representations for data and graphics would be expected.

References

Abelson, H. and G.J. Sussman, 1985, Structure and interpretation of computer programs (MIT Press, Cambridge, MA).

Bates, D. and J.M. Chambers, 1987, Statistical models as data structures, Statistical research report no. 42 (AT&T Bell Laboratories, Murray Hill, NJ).

Becker, R.A. and W.S. Cleveland, 1987, Brushing scatterplots, Technometrics 29, 127–142.

Borning, A.H. 1981, The programming language aspects of ThingLab, a constraint-oriented simulation laboratory, ACM Transactions on Programming Languages and Systems 3, 353–387.

Brodie, M.L., J. Mylopulos and J.W. Schmidt, eds., 1984, On conceptual modelling: Perspectives from artificial intelligence, data bases, and programming languages (Springer-Verlag, New York).

Goldberg, A. and D. Robson, 1983, SMALLTALK-80: The language and its implementation (Addison-Wesley, Reading, MA).

McDonald, J. A., 1986, Antelope: Data analysis with object-oriented programming and constraints, Proceedings of the ASA: Statistical computing section (American Statistical Association, Washington, DC).

McDonald, J.A. and J. Pedersen, 1986, Computing environments for data analysis, Part 3: Programming environments, Technical report no. 82 (University of Washington, Department of Statistics, Seattle, WA).

Oldford, R.W. and S.C. Peters, 1986, Object-oriented data representations for statistical data analysis, Compstat 1986, 301–306.

Oldford, R.W. and S.C. Peters, 1988, DINDE: Towards more sophisticated software environments for statistics, SIAM Journal for Scientific and Statistical Computing 9, 191–211.

Stefik, M. and D.G. Bobrow, 1985, Object-oriented programming: Themes and variations, The AI Magazine 5, 40–62.