Abstract data types as reusable software components: the case for twin ADTs

by K. Sikkel and J.C. van Vliet

Abstract data types (ADTs) are typical candidates for reusable software components. An important, although generally undecidable, question is how to design ADTs so as to make them truly reusable. In this paper, we present an elaborate example, showing that the most abstract ADT is not necessarily the most reusable one. Instead, we felt the need to supply ADTs reflecting alternative representations of one abstract notion. This leads to some general observations about the usability aspects of software components.

1 Introduction

Software reusability is one of the current buzzwords in software engineering. The demand for new applications far surpasses the ability of the collective workforce in our field. This gap between demand and supply is growing; there is a real need to dramatically increase software development productivity. Software reuse is one of the paths being explored in an attempt to achieve a considerable increase in productivity.

Reuse of software components has been successful in a few domains only, notably numerical mathematics and statistics. It proves to be very hard to apply the same techniques in other domains. Major stumbling blocks in trying to reuse components from an existing software base seem to be

- the naming problem: how the component is to be addressed.
- the usability problem: once a component is identified, how to assess its suitability for the problem at hand.

There is a strong correlation between those two problems. A component in the domain of mathematics, e.g., one that computes a sine function, is easy to find. Both the developer of the component and the reuser know this term. More importantly, they probably attach exactly the same semantics to this component. Thus, once it is found, we are fairly certain that the component offers the expected functionality.

In most domains, a strong link is not to be found between component name and semantics. Thus, searching for a component is likely to be problematic. Researchers try to overcome this problem by defining classification schemes [1] and/or (intelligent) retrieval facilities. Furthermore, we may define a 'conceptual distance' between components, which presents the possibility of retrieving components that are 'close to' a prime candidate found. Such techniques can only partly solve the reusability problem. Whether components are reusable is critically dependent on them incorporating useful, reusable concepts [2]. It has been noted [3] that usability of components is related to their perceived complexity.

Data abstraction is a well established technique for trying to achieve those aims. The concept 'Abstract Data Type' was developed in the mid-1970s [4, 5]. The idea of an abstract data type is to group together a set of data values and the set of operations on those values. ADTs offer excellent means of separating the functionality from the implementation of a data type. Direct access to the data is impossible. The only way to add, retrieve or manipulate the data is by means of operations, functions/procedures that interact with the ADT in a well defined manner. The functionality of the operations can be specified by (possibly formal) pre- and postconditions. The advantages of the separation of functionality and implementation are clear:

- the 'client', i.e., the user of the ADT, has a conceptually simple interface to deal with and should remain blissfully ignorant of the implementation details. Thus, the complexity as perceived by the user is decreased.
Examples of general-purpose ADTs that can be used for a wide variety of applications are usually rather small; the canonical example is a generalised stack. References 6 and 7 contain many other examples of such ADTs.

We argue that the separation of functionality and implementation becomes somewhat more problematic when the size and complexity of an ADT is scaled up. When a potential client is to judge whether a particular library ADT fits into their application, the interface specification may have to be supplemented with some understanding of what is going on inside the ADT. Implementations have pragmatic characteristics, such as range bounds on input parameters, precision (i.e. number of significant bits) of the calculated answer, what to do when a precondition is violated etc., that may influence their selection. McIlroy [8] identified 300 different conceivable sine function implementations. We can insist dogmatically that the client never ought to know anything about the implementation. However, that is denying the problem, rather than solving it. If we are to develop a deeper understanding of the practical usability of ADTs as software components, we should investigate

- in which way the implementation of an ADT influences usability for a particular application;
- how these implementation details can be abstracted to concepts that are understandable for the potential user.

We expect that, in general, the usability of components can be increased (and the perceived complexity decreased) when components match primitive concepts of the application domain. In Reference 9, we propose domain-oriented virtual machines (DOVMs) (collections of components that cover the primitives of a particular application domain) as a promising approach to reusability. Such a collection is tied to a certain application domain. Examples include a mathematical subroutine library. Lanegan and Grasso's collection of 'logic structures' [10] and the primitives of a fourth-generation language. At a more concrete level, the idea of a virtual machine also embodies a way of combining its facilities, such as a host language in the case of a subroutine library.

In order to test the DOVM hypothesis, we constructed a virtual machine for a domain in which considerable previous experience exists: that of dynamic systems. In the following, we highlight one particular outcome of this exercise, i.e. the need for twin ADTs. In trying to identify a useful collection of components for the domain of dynamic systems, we found that 'pure' ADT concept is sometimes unsatisfactory, in that it does not provide us with components at the right level of abstraction. Instead, we felt the need to supply the user with alternative implementations of the same ADT.

We provide a short introduction to the field of dynamic systems and present the ADTs that resulted from our effort in trying to identify a reusable collection of components for the domain. For a full exposition of the results of this exercise, see Reference 11. We also elaborate on the notion of twin ADTs or, more generally, avatars. Although the exercise called for dual representations of an ADT, generalisation to multiple representations is straightforward once the principle is recognised.

2 An elaborate example: dynamic systems

Some very elementary knowledge of control theory is required to understand the example. We introduce below the concepts needed; this can be passed over by readers with a basic knowledge of digital control theory. More details can be found in text books on modern control theory [12].

2.1 Some elements of control theory

Central to control theory is the notion of a linear dynamic system, or a system for short. The abstract graphical representation of a system is a box with an input and output arrow, as shown in Fig. 1.

The input x and output y are both functions of time. The output function is fully determined by the input. What happens inside the box is not important; it is a 'black box'. By definition, the relation between the input and output of a linear dynamic system can be described by a linear differential equation

\[ \frac{d^n y}{dt^n} + b_{n-1} \frac{d^{n-1} y}{dt^{n-1}} + \cdots + b_1 y = a_n \frac{d^n x}{dt^n} + \cdots + a_0 x \] (1)

For pragmatic reasons, control theorists write differential equations in a different format:

\[ H = \frac{y}{x} = \frac{a_n s^n + \cdots + a_1 s + a_0}{s^n + b_{n-1} s^{n-1} + \cdots + b_1 s + b_0} \] (2)

The symbol s denotes the differential operator d/dt. Crosswise multiplication of numerators and denominators in eqn. 2 and substitution of s by d/dt yields the conventional differential equation as given in eqn. 1. H is called the transfer function of a system. It is a rational function (i.e. the quotient of two polynomials) of the differential operator.
Linear dynamic systems can be combined in various ways, yielding a composite system that is also linear. In Fig. 2, the output of system 1 is directly fed into system 2. This combination yields another linear system, with the transfer function \( H = H_2H_1 \).

Another example of system construction is given in Fig. 3; the negative feedback loop. The purpose of a negative feedback loop is to keep the output \( y \) as close as possible to the input \( x \). Ideally, \( x = y \) and nothing happens. If for some reason \( x \neq y \), input \( e = x - y \) is fed into the inner box, causing \( y \) to move nearer to \( x \). In an unstable system, it is possible that a minor change in the value of \( x \) leads to an unbounded change of \( y \), and the equilibrium is never regained. Stability is one of the central issues in control theory.

Before we make things slightly more complicated, let us recapitulate. The input \( x \) and output \( y \) of a (linear dynamic) system are functions of time. Given a transfer function \( H \) and an input function \( x(t) \), the corresponding output function \( y(t) \) can be determined analytically. How this is done does not interest us here: the only thing to note is that a system is fully characterised by a transfer function, the ratio of two polynomials.

We have assumed \( x \) and \( y \) are real-valued functions of time, i.e. of the type \( R \rightarrow R \). We may also consider vector-valued functions \( x : R \rightarrow R^n \) and \( y : R \rightarrow R^m \). This is usually called a system with \( m \) inputs and \( n \) outputs, but that is mathematically identical to a system with a single vector-valued input and output. The transfer function of a system with \( m \) inputs and \( n \) outputs is an \( n \times m \) matrix of simple transfer functions, i.e. any system can be fully characterised by a matrix of ratios of polynomials. Composition of systems is similar to that of single-input single-output systems. By writing \( H_2H_1 \) in Fig. 2, we anticipated that matrix multiplication is not commutative. Fig. 3 only makes sense if \( m = n \). The transfer function of the system with an added feedback loop is \( \text{if} + H \text{i} \) \( H_1 \), with \( l \) the \( n \)-dimensional identity matrix.

As a second modification, we consider *discrete time*, rather than continuous time, with an input function \( x : Z \rightarrow R^n \) and an output function \( y : Z \rightarrow R^m \). The system does not continually monitor its input; the input is sampled at regular intervals. The output value accordingly is changed at the same intervals. Conversion of real-time input to a discrete-time input by means of sampling is a necessity if a digital computer is part of the system. In modern control systems, this is usually the case. Dealing with discrete-time, rather than real-time, systems involves mathematical details, but the principles do not change. Given an input function \( x(k) \) and a transfer function \( H \), the output function \( y(k) \) can be determined analytically. Note that functions over a discrete domain are usually called series. We can write \([x_k]_{k=-\infty}^{\infty}\) and \([y_k]_{k=-\infty}^{\infty}\) to denote the input and output series of a system.

Meanwhile, our notion of 'system' has expanded to a multidimensional discrete-time linear dynamic system. The behaviour can be modelled with a transfer function, which is, in fact, a special notation for a set of difference equations. There is, however, a completely different way to model such a system. At any moment \( k \) in the discrete time space, a system gets an input vector \( x_k \) and produces an output vector \( y_k \). Furthermore, the system has a state vector \( u_k \). The value of the state vector is determined by the history of the input, i.e. the series \( \ldots, x_{k-2}, x_{k-1}, x_k \). It encodes the information that is necessary to generate the appropriate output \( y_k \), based on the input series \( \ldots, x_{k-2}, x_{k-1}, x_k \). The behaviour of a system is described by

\[
\begin{align*}
\dot{u}_k &= Au_{k-1} + Bu_k \\
y_k &= Cu_k + Du_k
\end{align*}
\]

If \( l, m, n \) are the dimensions of the state vector, the input vector and the output vector, respectively, \( A \) is an \( l \times l \) matrix, \( B \) an \( l \times m \) matrix, \( C \) an \( n \times l \) matrix and \( D \) an \( n \times m \) matrix. Note that a system is now fully determined by four matrices of reals \( A, B, C \) and \( D \). This is called the state-space model or the ABCD model. We refer to the former model as the transfer function (TF) model.

For every TF model of a (multidimensional discrete-time linear dynamic) system, there is an equivalent ABCD model, and conversely, every ABCD model has an equivalent TF model. In practice, both models are used, because either is particularly suited for some type of application. In embedded systems, where a processor somewhere in a larger industrial control system calculates the output step by step based on real-time input, only the ABCD model is appropriate. At each step in time, only a limited number of vector operations need to be performed. On the other hand, if we want to determine the stability of a system model (before the real system is set into operation), the TF model is indispensable. Stability is not a property of the output at any particular moment in time, but of output functions as a whole; this is precisely what is modelled in the transfer function approach.

### 2.2 An ADT for linear dynamic systems

In Section 2.1, we introduced dynamic systems, in particular multidimensional discrete-time linear dynamic systems. This is the type of system for which we should develop a reusable abstract data type.

Which operations on systems should be supported by the ADT? We can draw up a preliminary list of requirements.

- **Define a system by giving a TF model.** We need to specify the number of inputs \( n \), the number of outputs \( m \) and up to \( n \times m \) rational functions for the transfer function matrix. It is not unusual that many elements of the transfer function matrix \( H \) are zero functions. Therefore, we only have to specify the non-zero rational functions \( h_{ij} \). Conversely, we should be able to retrieve a TF model...
from a system in a similar fashion. It should also be possible to change a system by replacing one of the $h_{ij}$. Note that we need to define (or reuse) a separate data type for rational functions, because that is the type of the parameters we used to communicate with the ADT.

- Define a system by giving an ABCD model. We need to specify the number of inputs ($m$) and outputs ($n$), as well as the dimension of the state vector ($l$). Furthermore, matrices $A$, $B$, $C$, and $D$ of the right dimensions must be given. It could be optional to include $D$. In most practical cases, $D$ is a zero matrix. Conversely, we should be able to retrieve an ABCD model from a system in a similar fashion. Note that a separate data type for matrices of reals would be convenient for communication with the ADT.

- Define a system through composition of existing systems. In Section 2.1, two examples of system composition were given. There are, in fact, a few more operations. Together, they form a ‘systems algebra’ with which any possible combination of systems can be constructed. This means that if a combination of linear dynamic systems can be wired together, such that the resulting composite system is also a linear dynamic system, a model of the composite system can be derived from models of the components.

- TF-type operations, e.g. use the transfer function. Transfer functions are used as functions in the complex plane (note, however, that coefficients are always real). The ADT should offer $n \times m$ functions $h_{ij}$; $C \rightarrow C$; or, alternatively, a function $H: C^n \times C^m$. This functionality could be located in the underlying abstract data type for rational functions.

- ABCD-type operations, e.g. ‘run’ a system. A system should be able to ‘fork off’ (or simulate) fresh state space machines, which can be used to generate the output corresponding to an arbitrary series of input vectors.

The above requirements of the ADT are taken from the design of a modular software package for controller design, based on Reference 13. A controlled system is a composite system containing (at least) a process and a controller, linked in a feedback loop. The process is (a model of) an industrial process, formulated as a dynamic system. The controller, usually implemented on a digital computer, is a dynamic system specifically designed to control the behaviour of the process.

There is no mechanical way to compute ‘the best’ controller for a given process; the controlled system may meet various (often conflicting) requirements to a certain extent. On the one hand, if the input is changed, the output should adapt to the new input reasonably fast. On the other hand, if the response is too fast, the system becomes unstable; rather than regaining a new equilibrium, the output oscillates ever more vehemently and the process is out of control. The right balance depends on the application at hand and is a matter of careful judgement by the systems engineer.

There are various methods available that highlight certain aspects of a designed system by means of graphs. In an interactive design session, the systems engineer defines a controller, checks relevant properties of the entire system, changes some parameters in the controller and so on. We do not discuss the graphics part of the design package, as it is beyond the scope of this paper. It should be clear, however, that computation of some graphs is based on the ABCD representation (e.g. plotting the system’s response to a characteristic input function), whereas for the computation of other graphs, the TF representation is needed (e.g. the Bode diagram, showing the amplitude ratio and phase change for sinusoidal input functions).

We have given an informal but not exhaustive list of operations on systems. It would not be difficult to give a more formal, complete interface specification along these lines. We know what the ADT should look like from the outside. It is not quite clear, however, how the ADT should be implemented. A central issue is the internal representation of a system. To keep things simple, we could maintain an internal data structure that closely resembles the TF representation (e.g. storing the coefficients of the transfer function polynomials in an appropriate manner). TF-type operations can then be implemented straightforwardly. As a consequence, implementation of ABCD-type operations is rather involved. A matrix-type representation must be derived from the primary TF-type data structure before the operation can be carried out. If a result has to be stored, it must be converted back to transfer functions first.

Alternatively, we could use four matrices as the internal data structure on which to store a system. For TF-type operations, the transfer functions can be calculated from the matrices and the results converted to matrices again: ABCD-operations can be implemented straightforwardly. Such conversions are hidden within the implementation and do not show up in the interface of the ADT. The user need not be concerned with these problems.

Unfortunately, this is not enough. Consider an application that is mainly interested in TF-type interface operations, where internal storage is of ABCD type. Conversion from ABCD to TF models is only possible by means of numerical approximation. Consequently, the ADT behaves very inefficiently and, worse, loses precision. A more sophisticated implementation of the ADT could keep a dual representation system. TF and ABCD forms are both stored when known, reducing the number of unnecessary conversions. While implementing the ADT some further decisions about the conversion strategy have to be taken. For example, is the product of a TF and an ABCD model to be in TF form, ABCD form, or both?

As a last alternative, if we cannot properly envisage the use that will be made of our ADT for dynamic systems, we can leave the decision as to choice and conversion of representations to the user of the ADT, i.e. the programmer of the client software. A further argument to underscore the eligibility of not completely hiding the internal representation is that engineers who devise or analyse dynamic systems think of a system in terms of its representation. If they are interested in the stability of a system, their notion of that system coincides with the representation most suited for answering that type of question, i.e. the TF model. If they want to find out how a system responds to a particular input series, the ABCD model is the more natural frame of thought. System engineers do not employ an abstraction more general than either of these two models.

At this point, we have to conclude that the internal representation of our ADT is relevant for the client software. This seems to be a sad fact, because it frustrates the very
idea of abstract data typing. An elegant solution for this case is presented below; in Section 3, we argue that this example is an instance of a more general class of data modelling problems.

2.3 An alternative; twin ADTs

In Section 2.2, we tried to design a generally usable ADT for linear dynamic systems. We found out that, from the interface alone, we cannot properly judge the usability of the ADT for a particular application. We have to know something about the internal representation, because conversion is a rather complicated operation with an associated loss of precision. We may extend the specification of the ADT with a specification of the conversion strategy, so as to allow the user to judge whether it is acceptable for a particular application. There is, however, a different approach, enhancing the reusability of the component.

Our problem was caused by over-abstraction; we want to know whether systems are internally represented by TF or ABCD models. Therefore, we should not hide the use of a particular representation inside an abstract data type. Consequently, we define the following components:

- an ADT for TF models, with operations
  - define, retrieve, change a TF model.
  - define a new model through composition of components.
- TF-type operations, e.g. use the transfer function.
- an ADT for ABCD models, with operations
  - define, retrieve, change an ABCD model.
  - define a new model through composition of components.
- ABCD-type operations, e.g. 'run' a system.
- conversion between TF and ABCD models.

The conversion routines can have access to the internal representation of either ADT. The implementation of the conversion routines, just like the implementations of both ADTs, can remain hidden from the client software. In this way, we create twin ADTs for linear dynamic systems (Fig. 4). The control over the type of model is passed on to the user.

The price to be paid for the increased control is slightly more work for the user, who has to administrate linear dynamic systems of different types and can call conversion functions when needed. The user must also be aware of the difference between TF and ABCD modelling, and remember which operations are defined on which type of model. In reality, this should never cause a problem. The application programmers can be expected to be knowledgeable about linear dynamic systems, otherwise they could not use the twin ADTs at all.

When a single, unified ADT for linear dynamic systems is required, for whatever purpose, it can be easily constructed using the twin ADTs. We need to define a conversion strategy and add an integrated interface offering a single data type that uses either twin ADT where applicable. This is illustrated in Fig. 5.

It is clear that the twin ADT approach enhances the reusability of our ADT for linear dynamic systems. The twin ADT offers maximum flexibility because the client software can decide about the representations used. Each twin ADT can be used on its own if only one type of model is needed. On the other hand, if we wish to hide the representations, a single, unified ADT is easily constructed on top of the twin ADTs.

3 Avatars

Let us review the issues raised in the previous Section, leaving our immediate concern of developing reusable abstract data types.

We have different models of a linear dynamic system or, to be more precise, a multidimensional discrete-time linear dynamic system. On the one hand, it can be regarded as a set of linear difference equations, denoted by a transfer function matrix. On the other hand, a system can be modelled as a state space machine that accepts input vectors, updates its state vector and produces output vectors. What really is a system? It is an abstract concept that can be represented by either model and perhaps many other models. We call these representations avatars, to avoid overloading the concept 'representation'. (Originally, an avatar is an incarnation, the form in which a Hindu god presents itself to humans. A god may appear in different avatars. In English, the word is used in the wider sense of 'form', 'manifestation'.) We have a notion of the abstract concept 'system', but if we want to describe, argue about, or retrieve properties of a system, we can only use one of its avatars.

A similar, more widely known example can be found in nuclear physics; an electron can be modelled both as a particle and as a wave. It has been empirically shown that an electron shows both particle and wave behaviour. This dilemma has been resolved by the view that (in our words) the electron-as-particle and the electron-as-wave are two avatars of one abstract notion electron.

We use the word avatar therefore in a technical sense for the representation of an object (or a class of objects) in a model, particularly when different types of modelling are possible. Two avatars of a particular linear dynamic system are its transfer function representation and its ABCD representation. These are not internal but external representations of the abstract type, because we can access an
Having introduced this new technical notion, induced by the peculiarities of linear dynamic systems as data types, we may wonder whether similar examples exist of abstract concepts with different avatars. The answer, surprisingly, is that it is quite easy to spot a few more. A simple example of a class of objects with two avatars is the complex plane. Every complex number can be identified by

- its Cartesian co-ordinates Re(z) and Im(z).
- its polar co-ordinates |z| and Arg(z).

Algebraic operations can be performed on complex numbers as a whole; but if we want to know the value of a complex number, we must use one of its avatars. In this case, we do not worry about internal representation, because conversion between Cartesian and polar co-ordinates is a trivial matter. It could be of interest, however, for applications that want to squeeze the last micro-seconds out of the available hardware.

In Reference 14, the complex plane example is treated within the context of object-oriented program construction. The solution provided by Meyer is to offer both avatars through the same class (i.e. abstract data type). The actual internal representation at any point in time is kept in a local variable that is hidden to the users. We could have applied a similar technique in our example. As argued above, there are good reasons for not doing so.

Having multiple possible representations of the same object raises some further issues: space overhead, conversion responsibility and maintenance. Before we discuss these issues, it is important to clarify the expected use of the different avatars. In our linear dynamics example, the twin ADTs offer disjoint sets of operations. For instance, the operation to run a system is offered by the ADT for ABCD models, but not by the ADT for TF models. In other examples, we may implement some of the operations for each of its representations, but we do not generally expect all operations to be provided by each representation. In that case, there seems to be no real need to have multiple representations to start with.

As a consequence we do not keep the various representations all of the time. The scheme we have in mind is graphically depicted in Fig. 6. In this scheme, there is only one valid representation at each point in time. In the example depicted, representation A is used to perform a sequence of operations. At a specified point in time, a switch to representation B is made. Subsequent operations use representation B, until a switch back to representation A is made. In this scheme, the problem of keeping representations consistent across operations does not exist.

In our twin ADT solution of the dynamic system example, the responsibility lies with the user as to which representation is used. When a single, unified ADT is offered, we may opt for several strategies. Conversion responsibility may then be left to the user or to the implementation, or some intermediate scheme may be employed. In the latter case, we may let the user suggest a change in representation; for example, to indicate that subsequent operations are better handled using the representation suggested.

The space overhead is caused by the fact that the different avatars each have to allocate space to represent the ADT. We may often temporarily discard either representation (as suggested in Fig. 6), but it is inevitable that both exist during the actual conversion between representations. Although we may conceive more complicated schemes in which one representation is gradually demolished while another representation is built up, this is felt to be rather impractical and sometimes even impossible.

Maintenance problems in this scheme are not unlike those found in other applications. If either representation changes, the corresponding ADT as well as the conversion routines must be adapted. If a change is to be effected in an operation provided by one of the ADTs only, only that ADT must be changed. If a change is required in an operation provided for two (or more) representations, the change is to be implemented twice as well (or more often). As argued above, this is expected to be the exception, rather than the general case. The type of maintenance problem for this special case, however, is similar to that involved in other software in multiple versions.

Another example of an abstract class with different avatars, somewhat similar to linear dynamic systems, is a class of polynomials. When designing the ADTs for linear dynamic systems, we needed a subtype for polynomials of complex numbers with real coefficients. These polynomials have the following avatars:

\( p(z) = a_0 + a_1 z + \cdots + a_m z^m \) (5)

\( p(z) = c(z - z_1) \cdots (z - z_m) \) (6)

Software Engineering Journal May 1992

Fig. 6 Conversion between representations

Object representing a system only through one of its avatars.
with the restriction that if \( \omega \) is a root of \( p(z) \), its complex conjugate, \( \bar{\omega} \) is also a root. This ensures real coefficients when the same polynomial is expressed in eqn. 3.

We may call eqn. 5 the additive and eqn. 6 the multiplicative avatar of a polynomial. Note that conversion from eqn. 6 to 5 is trivial (multiplication of the \( m \) factors), whereas conversion from eqn. 5 to eqn. 6 is much more difficult and leads to a loss of precision (numerical approximation of the roots of the polynomial). A simple conversion strategy can be based on the asymmetry of the conversion between the avatars. It is clearly superior to have the multiplicative form. Therefore, for example, if we want the product of multiplicative and an additive polynomial, the latter should be converted first.

If there is an optimal conversion strategy, independent of the application at hand, we could embed the two avatars into a single component, as in Fig. 5. Yet we do not need to hide the fact that this single component is based on twin ADTs. This increases the modularity, and hence the reusability of parts of the ADT (in case some application needs only one avatar). More importantly, anybody consciously using polynomials knows that eqns. 5 and 6 are the two avatars of a polynomial. An ADT should hide how the coefficients are stored (array or linked list, ascending or descending order), not the underlying theory of polynomials, on which the ADT is based.

In this respect, the main difference between polynomials with real coefficients and discrete-time multidimensional linear dynamic systems is in size and complexity. There is a similar asymmetry in the conversion between transfer functions and state space machines, allowing a reasonably good conversion strategy for most applications. Again, we suggest that twin ADTs or, more generally, ADTs with multiple representations, are a useful concept in the construction of reusable software components for data types with more than one avatar.

5 Acknowledgment

The authors would like to thank the referees for their comments.

6 References


The paper was first received on 10 April 1990 and in final revised form 10 September 1991.