APPLICATION AND DATABASE DESIGN -- PUTTING IT OFF

Christopher J. Shaw
TRILLIUM
3770 Highland Avenue, Suite 208
Manhattan Beach, CA 90266
(213) 545-8300

There seems to be two major schools of thought about designing database application systems:

1. System design is **mandatory**, say the methodologers. How can you ever build a system unless you first determine in detail what it should do?

2. System design is **impossible**, say the prototypists. How can you ever build a system if you must first determine in detail what it should do?

In this article, I outline a deferred approach to application and database design that may be acceptable to both camps. It's intended primarily for use with so-called Fourth-Generation Languages, where applications are defined in terms of screens, records, fields, and reports, and, where absolutely necessary, procedures.

My approach is based on the familiar ideas of **objects**, that exist over time, and **events**, which happen from time to time, affecting them. Objects and events are what such systems are all about, and a design based on these concepts is likely to be more comprehensible to users than one based on data flows, or data relationships, or on object states (records) and changes of state (record updating).

Later in this article, I explore the various system roles played by relationships among and between objects and events, and argue that predefined, stored relationships are primarily needed to reduce processing requirements to tolerable levels. I also discuss the concept of object state, and how to put off thinking about this difficult and debilitating topic as long as possible. Object states also serve to reduce processing times, but they primarily do the complementary job of reducing storage requirements by summarizing events.

System Design Just as there are two schools of thought about system design, approaches to actually doing it often fall into two very broad categories:

1. **Function First:** determine the system's functions, then figure out what data is needed to support them.

2. **Data First:** design a good data model for your application, and then specify the system's functions in terms of it.

Traditional application system design focuses on function before data, typically in the following three steps:

1. Define inputs and outputs.
2. Define processing functions.
3. Define database requirements.

Author's current address is: Chris Shaw, Ashton-Tate, 20101 Hamilton Avenue, Torrance, California 90502-1319.

More modern, data flow design techniques (as reviewed by Chapin [2]) add the wrinkle of top-down decomposition, as follows:

1. Who supplies the data?
2. Who uses the data?
3. Trace the flow from supplier to user.
4. Insert processes whenever changes occur in the data.
5. Enter identified data elements into a data dictionary.
6. Do a data flow analysis for each process of step 4.

One problem with this structured, but still function-oriented approach is that it typically requires specifying and respecifying the system in a variety of different languages:

1. System requirements are stated in English.
2. Data flows are specified in data flow diagrams.
3. Control flow is specified in structure charts.
4. Detailed design is specified in a program design language.
5. Programs are coded in a programming language.

Producing these multiple specifications is laborious, slow, and error prone. Moreover, it requires trained data processing specialists, and does not encourage user involvement.

Basing a system design on function is only sensible when the system's functions are well known, and not subject to much change. This is often just not so.

Data Design The data-before-function approach to system design aims at a database capable of supporting a wide variety of functions. Indeed, the very idea of a database implies support for different applications, some of them not yet known.

According to William Kent [5], there are just two central questions in logical data design:

1. What **records** should be specified?
2. What **fields** should those records have?

Again, approaches to answering these questions usually fall into two categories:

1. **Fields-first**: start with the fields, determine their functional interdependencies, and normalize them into records.

2. **Records-first**: start with the records, and then determine what fields they should contain.

Lien [6] provides a good overview of the fields-first, normalization way of database design, and Curtice and Jones [3] give a very good account of the records-first approach, as does Sweet [7], who convincingly argues that database functions (such as record addition, deletion, modification, and navigation) can be pretty much standardized.

Unfortunately, whichever way you pursue it, data design is a demanding and essentially obscure discipline. Just try explaining to a user why Boyce-Codd normal form is desirable, or better yet, the meanings of the many dinky little tables you often get by actually doing this normalization! And because a database is usually a snapshot of a single state of the reality it models, the dynamics of the situation are necessarily modeled by database updating transactions.

Functional design and data design are both very difficult -- books can and have been written about how to do them. So it's natural to ask: Are they necessary? Can they be avoided? Or, if not, can they at least be deferred?

Putting Off Design Is it possible to build a useful database application system without first considering what functions it must perform and what data elements it must store? You bet it is, as I hope shortly to explain. And what uses might such a system have? At the very least, simply this: to help users answer the following questions:

- What should the system do?
- What must the system remember?

To avoid asking these difficult and threatening questions up front, when the answers would have ramifications and consequences even the most sophisticated users (and designers) would find hard to predict, we begin instead by asking the much less threatening though equally consequential question,

- What is the system all about?

Since systems, at least of the kind we are considering, are all about objects, and events, this last question, breaks down into the following subquestions:

1. **What is the system all about?**
   1.1 What are the objects?
   1.2 How are they identified?
   1.3 What are the events?
   1.4 What do we know about them? In particular,
      1.4.1 What objects do they involve?
      1.4.2 What other descriptive properties do they have?

Note that we're not asking about database objects or system events, but about real-world objects and events, outside of the system. Thus, what we have in the answers to these questions is a model of (some aspects of) the real world, not a system or database design. And, it is a generic model, since it is applicable to all possible systems involving the same objects and events.

We can ignore this fact -- that we have a generic, real-world model, not a system or database design -- if we wish, however. By using a Fourth-Generation Language, we can build a prototype system based on the answers to these questions, storing both object identifications and event descriptions as database records. The event records would naturally include the date-time of their occurrence or, if not, their receipt by the system.

Such a prototype, once the database has been experimentally populated, has the advantage that it contains the answers to all the questions the user could reasonably ask about the recorded objects and events. No potentially useful functions have been excluded by the system designer, except those involving objects or events left out of the model.

The disadvantages of such a prototype are equally striking. Although it holds the answers to all reasonable questions,

- Some questions may require complex queries and are therefore difficult to ask.

- Other questions may require a great deal of processing and are therefore difficult to answer.

Nevertheless, by limiting the amount of data involved, such a prototype can serve as an excellent experimental vehicle to help the users answer the system designer's real questions, which can now be stated, in terms of the system's objects and events, as follows:

2. **What should the system do** (and when should it do it)?
   2.1 What are the system's functions? (These can always be specified in terms of relationships among system objects and events.)
   2.2 How are these relationships to be implemented?

3. **What must the system remember** (and for how long)?
   3.1 What are the object's states?
   3.2 How are these states determined and stored?
   3.3 How long should event records be kept?

These three steps outline a minimal, deferred approach to system design. *First*, determine the interesting objects and events, and maybe build a prototype system based on them. *Second*, determine the interesting object/event relationships. *Third*, determine the interesting object states.

I'll discuss some of the issues involved in actually doing this next.

Objects Exist Objects are things that can be viewed as existing and changing over time, as having a life-span and, in particular, a life history of on-going changes. Objects do not necessarily imply individual, physical existence. They can be activities, places, organizations, systems, or collections of things.

Objects are the system's subject matter, and should be chosen with care. Although you should include all the objects (and events) the user cares about that fit within the system's agreed-upon scope, certain rules about objects are worth considering:

1. Occam's rule -- don't multiply them unnecessarily, if the distinctions aren't likely to be useful. In particular, candidate objects should be excluded if they:

   - have no appreciable life-span, or undergo no relevant changes;

   - have the same life-history of events as another object.

   Thus, a group or category of objects should be included as a separate object only if there are events that involve the entire group or category, and it has an interesting life history separate from that of its members or instances.

2. Don't have separate object types if one type of object can change into another, and their continuing identity is important. Combine them into a single type. In particular, objects that pertain to just part of the life-history of another object or only to certain instances of another object are questionable. For example, if a caterpillar can subsequently become a butterfly, and we are interested in this transformation, then we should opt for a single object type to include them both.

3. Do have separate object types if it's clear their data requirements or life histories are different. For example, if our system must handle payroll, customers and employees are best kept as separate object types. On the other hand, the fact that maternity leave is an event that only applies to female employees can probably be handled without having separate male and female employee types.

4. Don't treat events or event records as objects. For example: a customer ordering goods is an event; the actual order is a record of that event. On the other hand, processes, which consist of possibly conditional or repetitive sequences of events, can themselves be interesting objects. Thus, the process of fulfilling an order could very well be treated as an object. It depends on what's important to the users, and whether they consider orders as separate, identifiable entities. Another example: Is a shipment an object or an event? It depends. When a customer calls up and asks, "Where's my order?" do you want to answer, "It's been shipped," (shipment as event), or "It's on a flatcar heading North from Nashville" (shipment as object).

The crucial test for objects is this: are the users interested in its history?

The notion of object or entity is central to most semantic data models, which support such semantically rich relationships as classification, generalization, specialization, and aggregation. Although real-world objects have types and categories, and complex objects have structure and components (which are also objects), in current database systems, these aspects are often best dealt with as relationships among objects. Objects in such systems can only belong to a single type, but can belong to many different relationships.

Events Happen Events are the things that happen that cause objects to change. They can be viewed as occurring instantaneously, at particular moments of time. Events don't have a life-span or life history and, once they have occurred, they don't undergo change -- except, perhaps, to be corrected.

Events always involve objects, and rarely need to be otherwise individually identified, except by date and time. In general, a single event may involve (and cause changes in) several objects. An event that affects a variable number of objects can usually be represented either as a set of events, or as a single event involving a collective object (or several collective objects), if the database management system you are using doesn't provide for repeating values.

Again, Occam's rule applies, and a candidate event should be excluded from the system if it is:

- Not economically detectable, or cannot somehow be reported to the system.
- Really an object state or relationship, with a duration. For example, borrowing a sum of money is an event; owing money is a state or relationship.
- A synonym for another event. In particular, two events that always happen together (or that always get reported together) can usually be replaced by a single event. For example, an employee is born, and is later hired. The hiring event record can include date of birth, so there may be no need to provide for a separate birth type of event.
- A process, composed of subevents. Of course, all events are processes if you expand the time scale enough. The issue, as always, depends on the users' view. If their only concern is whether or not the process has occurred, it's an event. If they're concerned with how far along the process is, then it's probably an object, with a life history of events, or maybe just a relationship among the events.

The use of events as a primitive concept is strangely rare in current work on data modeling, even where time is a focus of concern. One notable exception is Jackson [4,1], whose system development method is soundly based on objects and events. Indeed, this present work comes from applying the initial steps of Jackson System Development to Fourth-Generation application development environments.

Relationships Provide Functionality After we've determined what objects and events the system is concerned with, we must now consider its purpose, the functions it is supposed to perform. These are typically defined in terms of outputs the system should produce when certain situations or combinations or events occur.

It is important to realize that the functions of a data-based information system can always be expressed in terms of relationships -- relationships between and among objects and events.

In a library system, for example, do we wish a list of overdue books and their borrowers? That's a relation between books and readers, involving borrow and return events. A list of books whose copies are all out on loan? That's another relation, involving book editions, book copies, and borrow and return events

Systems often exist to help enforce rules, such as: No reader can have more than six books out on loan. Rules are often intended to eliminate or minimize some specific relationship. For example, the rule that no reader can keep a book on loan more then three weeks is intended to minimize the relationship of books borrowed more than three weeks ago and not yet returned.

A system design that incorporates such rules as stored relationships, cast into the concrete of data structures, is sure to disappoint when the organization liberalizes or changes its policy. Besides, rules like these are made to be broken, and systems need to be able to keep track of exceptions as well as conventional cases.

Relationships involving objects and events are much like objects themselves, in that they typically exist over periods of time. Such relationships may therefore also be considered as objects, but only if they participate in events of their own. Nevertheless, by storing relationships in our database, as well as objects and events, we can often reduce the processing needed to provide many of the system's functions.

For example, if we need to know which books a given reader has out on loan, we could determine this relationship from the reader's history of borrow and return events, but it might be easier just to keep a list, for each reader, of borrowed books. Books would then be added to this list when borrowed, and removed from it when returned.

One serious problem always arises, however, whenever we choose to store any information about an object's current relationships. Since that information is redundant, in that it duplicates what can be computed from the object's life history of events, it is always possible for the current relational information to be inconsistent with the life history. For example, a reader's life history may include 125 borrows and 123 returns, yet that reader's list of books on loan might, for a variety of all too likely reasons, contain four entries, or even two incorrect entries.

Another equally serious problem arises when events are reported to and processed by the system out of turn. If, somehow, a return event is processed before the associated borrow event, the system must be programmed to deal with its failure to remove a nonexistent entry from some reader's list of books on loan.

These kinds of problems make database systems based on maintaining stored relationships very difficult to design and manage. Users want to get as much benefit as possible from their data, but they're usually willing to cut back if certain relationships prove too hard or too expensive to maintain. Unless your users really need the performance provided by stored relationships, you might be well advised to stick to objects and events.

States Summarize Objects have states. That is, during their life-span, they continually change from one state to another as events affect them.

What constitutes the state of an object? An object's state is represented in a database by the totality of stored information relating to it. Aside from event records, and the stored relationships in which the object participates, this status information can be grouped into four main categories:

1. A state pointer, indicating where the object is in its life-cycle of events. (Given that the life history of a class of objects can be represented as a hierarchical structure of sequences, selections, and iterations of events, the state pointer can serve to indicate what event or events are expected or permitted next.)

2. Properties of the most recent events, for example, address changes.

3. Summaries of past events, such as account balances.

4. References to any predefined (but not necessarily stored) relationships to which the object may belong; for example, customers with overdue accounts.

Note that there is no fundamental difference between states and stored relationships. The only difference is how they're implemented. State information is stored as part of the object record. Relationship information is stored apart from the object record.

In principal we can always compute an object's current state from the system's life history of events, and maybe even just from the *object's* life history of events. But unfortunately this is not always convenient.

For example, if we want to know how many books a given reader has out on loan, we could add up all his borrow events, and subtract all his return events. But it may be easier to keep this number as part of the reader's state, and increment it for each borrow and decrement it for each return as these events are reported to the system.

A system that has to sort through records of a library's entire past history of borrow and return events before it will accept a new borrow transaction might even work for a small, fairly new library, but not for long. And even though the cost of storage grows cheaper seemingly by the week, a system that is only interested in the current state of a great many highly volatile objects could find the cost of storing a complete event history excessive.

Consequently, the twin strategies of updating stored object states and purging old event records are usually necessary evils in most application system development projects. Nevertheless, you should recognize that these are momentous decisions, to employ these strategies. While they may make the system economically feasible, they also introduce fundamental limitations and difficulties into the system design.

Summarizing event data in variable-state object records means that we necessarily eliminate many of the relations or functions we might otherwise be able to compute. For example, summarizing purchase and payment events into an account balance means that when the user asks, "Don't just tell me how much this customer currently owes, tell me how much business he's done with us over the last few years," the system can have no way of responding.

Any information we lose by purging event records can always be included in object states or stored relationships if we can foresee the need for it. But, of course, users can't easily predict all the requests they'd eventually like to make of their systems. So beware of eliminating event data without making sure the users understand the kinds of questions the system will no longer be able to answer.

As we have already seen with regard to stored relationships, summarizing event data in variable-state object records also introduces the possibility of object states that are incompatible with the events. Another, more fundamental difficulty arises when an event affects the state of two or more objects (or relationships). For example, a reader returning a book. Care must be taken to update the states of both objects and of any relevant relationships. If the transaction fails to complete for any reason (power failure, hardware failure, software failure), and only updates some of the entities, then the database will be left in an inconsistent, indeed, erroneous, condition. Special care is always needed to minimize such problems, and often this is the system developer's responsibility.

Summary I have, in this article, outlined an approach to designing database application systems suitable for use with Fourth-Generation application development tools, and especially suitable for building prototypes, for establishing system requirements.

The basic idea is to defer all consideration of what the system is supposed to do, and what the system is supposed to remember, until a model of the system's application world is developed. This model employs just two, primitive concepts: objects, which exist, and events, which happen to them. Such an object/event model is applicable to all possible systems involving the same objects and events, yet it can be implemented quite directly, to serve as a generic system prototype, by storing records containing object identifications and event descriptions in a database and using a query system to retrieve information on the relationships among them.

Such a prototype is also helpful in demonstrating, to the users, the trade-offs that arise when event reports are discarded in favor of stored relationships and object states. While the benefits of this are faster processing and reduced storage, the costs include the added complexities of transaction processing, the liklihood of reduced fidelity to the underlying event history, and limited functionality with regard to the history and dynamics of things. An object/event-based prototype allows these trade-offs to be made on a case-by-case basis, with the users' knowledge and concurrence.

References

[1] John R. Cameron. **JSP & JSD: The Jackson Approach to Software Development,** IEEE Computer Society Press, Silver Spring, MD, 257 pages, 1983.

[2] Ned Chapin. **Structured Analysis And Structured Design: An Overview.** In Cotterman, et al (editors), <u>Systems Analysis and Design: A foundation for the 1980's,</u> pages 199-212, North Holland, 1981.

[3] Robert M. Curtice and Paul E. Jones. **Logical Data Base Design.** Van Nostrand Reinhold Co., 227 pages. 1982

[4] Michael Jackson. **System Development**. Prentice/Hall International, Englewood Cliffs, NJ, 418 pages, 1983.

[5] W. Kent. **A Realistic Look at Data**. In Database Engineering, vol. 7, no. 4, pages 22-27, December 1984.

[6] Y.E. Lien. **Relational Database Design**. In S. Bing Yao (editor), Principles of Database Design, Vol. 1, Logical Organizations, pages 211-254, Prentice Hall, 1985.

[7] Frank Sweet. **Building Database Applications**. Boxes and Arrows Publishing, Jacksonville, FL, 258 pages, 1986.