Summary fields are useful because they enable one program to do the summarization work for many others.

the diocesan totals. If each were to compute the sum on its own, it would have to read all the parishes in every diocese to get the numbers it needs. Each would be more complex, thereby taking longer to write, compile, debug, test, and maintain. Each would run longer.

By using summary fields, only one program (the one that maintains the lowest-level record—the parish) does the time-consuming work. Other programs obtain the data right out of the higher-level records. This makes them simpler and faster. It also requires a bit more disk space, but 3350 space rents for about $2 per megabyte per month and isn't a serious factor.

Summary fields are risky because they are redundant. It's rather like keeping old balance, debits, credits, and new balance on the same record (any three are enough to do the job, all four is overkill). Being redundant, they can (and most likely will) become inconsistent. After a year, it is a safe bet that 10% of the records on file will carry summary totals that do not match the sum of their details. The causes are legion:

• User procedural error. A procedure says once summaries are computed, users must not modify details. But no procedure is followed perfectly by everyone all the time.

• Program bug. A program updates three parish records, goes to update the diocese record, and blows away with a data exception. Depending on how the files are managed, the summary totals may no longer match the details.

• Program maintenance. A programmer enhances an on-line inquiry program, enabling the user to update diocese records. The specs don't warn that its summary fields must be protected. They're put on the screen unprotected, and users promptly start changing them. (Specs? What specs?)

• System crash. An on-line program updates a parish record, gets ready to read the diocese record so it can increment the summary fields, and CICS crashes.

But all the causes are irrelevant. The point is that when anything goes wrong, the diocese records will be bad and the fact may never be detected.

Again, there is no absolute answer. If detail records are added or modified online but summary totals are only needed at the end of the month, we don't need the redundant summary fields at all. Why force every update program to read higher-level records, increment their totals, and rewrite them whenever a detail changes?

But if detail records are keyed in batches, while summaries are inquired online, why force every inquiry to read all the details and add them up on its own when the batch update could have done it once and for all?

Our best bet is to decide where our situation lies, somewhere between these two extremes. If we feel we need summary fields, we include them in our design. We are aware of the risk though, and make sure our users understand that inconsistencies will arise and they must find some way of detecting and correcting them.

SOME RECURRING PATTERNS

Now, in part nine ("The Identifier and Its Name") let's look into recurring patterns that appear in many database designs.

Designing databases can be curiously repetitive work. It's not only that after we've built a dozen inventory systems the thirteenth seems somehow familiar. Even working on an application where we have no prior experience we may feel a sense of déjà vu. Designs seem to take on their own lives. Useful ones appear again and again, heedless of application. Think of examples we've met: that neat way of handling a partial-name alphabetic search for a client's personnel file, reincarnated months later to handle another's vendor file. The audit trails we sketched to track money in an account are remarkably like those we later used to keep track of goods in a warehouse. Some techniques, indeed, are so reliable and widely useful that we can call them patterns in database design.

For the next couple of sections, let me show you five of my favorite patterns. They range in complexity from the simple identifier and its name examined today to the strange, labyrinthine, self-related record we'll meet in part 13 of this series.

Some are so widely applicable to information storage problems that they apply even to paper files. Others are limited to database management systems. What they have in common is that we've encountered them so often, in so many guises, they now seem like old friends. We can rely on them to do the job. Often, they will warn us when we miss something vital. More important, they help us listen to our user with productive skepticism. But more about this later.

The five patterns are
• the identifier and its name,
• the past event,
• the future event,
• the template, and
• the self-related record.

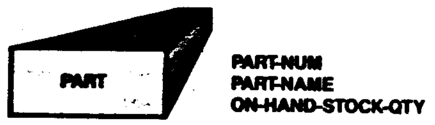
To make them tangible, we'll use them to design a sample application. Starting with a statement of system scope, we shall apply each, in turn, to derive a finished conceptual database design.

Our application is an imaginary maintenance equipment spare parts inventory system. The first order of business is to come up with an unpronounceable acronym, and our system is no exception. We'll call it MESPIS, and its threefold scope is to
• keep track of the on-hand stock quantity of each item in our firm's spare parts warehouse;
• produce a report when it is time to re-order a spare part, based on its stock quantity being too low; and
• identify, for each part, those pieces of equipment in which it is used.

Our first pattern is the identifier and its name. Every entity (everything about which we'll store business data) carries both identification fields and data fields. ID fields identify, data fields describe. ID fields tell our client's staff as well as our programs which specific occurrence of the entity is at hand. Data fields describe attributes or characteristics of the entity.

As we saw in "Objects and Events," (see part three of the series, Sept. 15, p. 152) entities come in two groups, volatile events and involatile objects. The pattern simply tells us that objects should always carry at least one field of each type: an entity identifier (an ID-number or keyfield) and a name (or description). Without these two fields, we wouldn't have a viable entity at all.

Since our system deals with objects (spare parts), we begin by drawing a box labeled "part" and know that the record must have an identification number of some sort and a name.



Of course, since the scope statement spoke of stock quantity, we include this field as well.

Now and again we'll run across a client who affirms that a proposed involatile entity needs no name or description. We include one nonetheless (productive skepticism). If we neglect to do so, we'd probably come back in six months and find them using address or location or some other fields to store the information. To keep the other field usable, we would then retrofit the record and add description after all. So, we might as well do it now and get it over with.

Next time, we'll continue designing the MESPIS database by examining past and future events. ◉

Frank Sweet is corporate manager of data administration for the Charter Co., Jacksonville, Fla.