

Lessons Learned from Developing Small Databases

As a business develops and times and personnel change a database needs to evolve

By Jackie Grubb (jackie@plumsuite.com)

I've been hooked on developing small customized databases for over 20 years, since back in the days of DOS-based PCs.

From my very first project in this line, I've had a chance to see how much even a machine that was fairly primitive, by today's standards, could accomplish. The *aha!* moment for me came when the COBOL programmer I was working with in the final phase of that project commented to me that the database I'd developed (using a computer he considered to be a mere toy) would have taken five or six COBOL programmers three years to accomplish.

In the years since, despite vast improvements in off-the-shelf software as well as hardware, I've seen dramatic evidence of how the right customized database can help a small business or nonprofit organization accomplish much more, with much less staff.

The first project I undertook – a system to automate payments for a publishing company's remote freelancers – involved just two people: me as the developer and a department head who was able to articulate and focus on the department's needs. It took three months to develop, including needs analysis, prototyping, and implementation, and then ended up saving half a clerical position. The program required some tweaking during its first year in use. But after that, it needed no further work, except when the accounting folks decided to change their account numbering. It continued to do its job until the department went out of existence.

In the years since, I have developed probably more than 30 small customized databases focused mostly on optimizing clerical work. I have seen some patterns in the process, especially in the way a program can evolve over the years.

Recently, at a music school I have worked with for over 10 years, two new high-level employees asked me to review with them the database I have developed to generate teacher schedules, manage the mailing list, track fundraising appeals and calculate lesson fees which are submitted automatically to their accounting software. Like many of my projects, it has been through a series of development phases as the program proved itself and additional functionality was desired. Sometimes three or more months would go by when I did nothing on the program – but then there would be a new need and I would work for a month or two on an upgrade.

As I was showing the features to the new employees, I found myself explaining how the program has evolved – and what conditions had prompted some of the features. The review pointed up to me that the database program retained several artifacts of conditions that no longer exist. It also highlighted how the program had evolved as conditions changed for the school.

When I first started working with the school, the office staff consisted of an executive director, a part-time registrar, and a part-time bookkeeper. The executive director was very well organized, with a firm understanding of the needs of the school and an ability to articulate them. Lessons were held late afternoon and early evening at the local middle school. The music school had around 30 teachers and about 500 students.

About five years ago, the school moved into its own building. It has since blossomed and now has an office staff of four full-time and several part-time people. The teaching staff and student load have both nearly doubled. The database is doing a good job for them. I recently visited another music school serving a similar number of students. Without a workable database, they were accomplishing much less, despite an office staff of at least 14.

Since I started working with "my" music school, the executive director has retired, and there has been a complete staff turnover. As new staff have used the program and looked at it with fresh eyes, they have seen new ways it could be enhanced and have also turned up previously undiscovered bugs.

Different people try different things with a program, and it sometimes takes quite a long time before a particular combination of conditions will highlight a problem. Some conditions are cyclical and occur only once a year (or even once every four years, as in leap year). An example of this occurred when we automated the entry of the dollar amounts of the lesson fees. Before, the one entering data had looked at a paper chart posted on the wall and picked out the amount to be changed for, say fifteen 45-minute piano lessons.

It appeared that automating the process would make the entry more accurate and eliminate lookup time. So we incorporated a lesson-price lookup table to allow the program to calculate and fill in the correct amount automatically when the entry operator chose the standard fifteen 30-, 45-, or 60-minute lessons. Since the prices changed each summer when they published their catalog for the following year, we planned for the prices to be changed once a year.

What nobody thought of when we instituted the change early one fall was that there would be a period when they would be entering lessons for the summer session using the old prices while also needing to enter lessons for fall using the new prices. That issue did not emerge and get remedied until late the following spring.

So, I learned yet another lesson about the cyclical process, which I will remember, I hope, the next time a similar situation emerges. Part of what a seasoned developer brings to a project is all those lessons learned – things you would never think of if you had not been through the process. I have learned that it takes at least a year of a new program being used before all of the cyclical issues will be addressed. There will always be some little thing that nobody has thought of.

Staff turnover always uncovers new needs and new ways of working with the software. I have been through complete staff turnover with one or two of my other clients now and have had upgrade requests come as a result. Printed reports are frequently the result of someone's desire to look at data in a particular way – and sometimes these get changed as staff come and go.

Another issue related to staff turnover has to do with frequent change of the people inputting the data. In such situations the program needs to be tightly set up so as to eliminate the possibility of entering incorrect data. Sometimes temporary help is needed to get required data into the system. In these situations, there is little time for training, so the program needs to make the way the data is to be entered very obvious.

In the case of the music-lesson entry, for example, the entry operator is given a pick-list from which to choose – among the 30-, 45-, or 60-minute lessons or a specific ensemble or class. Limiting choice to a pick-list helps to preserve data integrity by eliminating the possibility of the data being entered according to the operator's whim. Providing for the automatic generation of the dollar amount has also helped in the high clerical turnover situation.

I get my best ideas on how to simplify and "bulletproof" data entry by sitting next to and watching someone use the program to enter real data. It helps me to see the problems through a typical user's eyes.

So, some lessons learned, that I believe apply to most any size custom software project, are:

- Changing conditions and staff may demand software changes.
- It takes at least a year, sometimes longer, for the majority of program bugs and problems to be identified.
- As a business grows the software requirements will probably grow as well.
- A 'bulletproofing' process is required if data is to be entered by untrained personnel