Whatever happened to 'database table refactoring'? Around ten years ago, several books were published on the topic, but evolutionary database design never really caught on as an exciting way of developing relational databases.

The idea behind evolutionary database design was that you could use a number of 'refactoring' techniques to develop the database iteratively, in parallel with the evolutionary design of the rest of your system. You just created tables that allowed you to 'persist' the objects. As your understanding of the business domain evolved, you just altered the objects and the database tables together in step.

The proponents of evolutionary database design had the set idea that it was intrinsically difficult to change a relational database. Their mission became one of writing books and articles to explain how one could do so. The NoSQL movement even took pride in providing a more easily-changed 'schema-less' database.

Actually, it isn't too hard to change your relational database design nowadays. The real reason that this approach isn't generally suitable is more subtle. In a large proportion of organizations, you are legally obliged to know about your data, and deal with it properly. Governance has to precede application development. The data in most enterprises has to be understood up-front because some of it is subject to a growing legislative framework and to industry codes of practice. Sometimes you don't own it, sometimes you can't retain it; usually you must keep it secure, and control access to it: It has to be audited in special ways. If you get it badly wrong, your enterprise will be adversely affected, even occasionally destroyed.

Processes can be changed with comparative ease, and can be 'evolved'. Application developers therefore have the liberty to be creative in the way that they can automate a business process. The database designer often, by contrast, is restricted by the nature of the data, and by the constraints inherent in the data. If, for example, some of the data needs to be ring-fenced into a strict security, access, high-availability or auditing regime, and you haven't provided it, you are likely to be facing delays and budget problems far worse than the relatively minor task of table-refactoring. If you have a design that doesn't adequately fit the data, or that allows wild interdependencies, then you'll have all the agility of a fly in treacle. If you have Business Intelligence applications using the data as well as one or more applications, then gridlock can happen very quickly if the teams cannot closely-coordinate their work.

Neither sophisticated 'refactoring' technology nor clever teamwork can easily remove the need for planning and careful design of databases, whatever their flavor. They can be used to rectify mistakes, of course: if they are used to avoid 'design-up-front', then refactoring techniques can postpone the inevitable problems, but they can't avoid them.