What Not How

The Business Rules Approach to Application Development

C.J. Date

What Not How

The Business Rules Approach to Application Development

Excerpt from the upcoming book by C.J. Date.

chapter 1

What’s the Problem?

When they first began to appear, in the early 1950s or so, computers were very hard to use–they required very specialized skills, and you really had to be a computer technician in order to use them at all; originally, in fact, you probably had to be a hardware engineer. Over time, however, computer systems have become much more "user-friendly" and easy to use, thanks to a continual *raising of the level of abstraction*: so much so, in fact, that now you can use a computer effectively even if you have almost no knowledge of how its internals work at all (much as you can drive a car effectively even if you don't know what goes on under the hood). Here are a few familiar examples of that "raising of the level of abstraction" that have taken place over the years:

- `1GLs f 2GLs f 3GLs f 4GLs`

  Programming languages have evolved through several "generations," from first generation languages (1GLs) to at least a fourth generation (4GLs). Just to remind you: 1GLs were machine languages; 2GLs were assembler languages; 3GLs were the so-called "high-level" languages(COBOL, Fortran, and the rest); 4GLs were various proprietary languages, such as FOCUS from Information Builders, Inc. Some people regard SQL as a 4GL [15].*

* Throughout this book, numbers in square brackets refer to publications listed in the appendix.

- `Sequential files f indexed (ISAM) files f hierarchic and network databases f SQL tables`

  Over the years, more and more of the details of storing and managing data have been taken over by the system; in a word, they've been automated. Nowadays, it's the system, not the user, that's responsible for finding data as and when required (and finding it fast); it's the system, not the user, that's responsible for recovering data in the event of failure; it's the system, not the user, that's responsible for protecting data from concurrent update; and so on.

- Specialized languages and interfaces: for example, RPG, SQL, QBE, "visual programming" (QBF, ABF), spreadsheets, ...

  This one is more or less self-explanatory. However, I'd like to elaborate briefly on QBF and ABF, because they're directly relevant to some of the ideas we'll be examining in the next few chapters. QBF–*Query By Forms*–allows you to do database queries and updates* by making simple entries in a form on the screen. ABF–*Applications By Forms*–allows you to develop applications in the same kind of way, and those applications in turn also use on-screen forms as the interface with the user. Note: QBF and ABF both grew out of work originally done in the early 1980s at the University of California at Berkeley [25]; they were first commercialized in the Ingres product, originally from Relational Technology Inc., now–under the name Ingres II–from Computer Associates International Inc.

* In accordance with normal usage, this book uses the term "update" generically to include all three of the familiar operators INSERT, DELETE, and UPDATE.

Let me add a word on spreadsheets, too. Spreadsheets raised the level of abstraction, in their particular field of application, by getting away from writing programs *entirely* (nobody today would use Fortran to write an application to perform spreadsheet-style processing). There's a very direct parallel here with business rule systems, as we'll see.

In a nutshell, then, it should be clear that the historical trend has clearly always been away from *procedural* and toward *declarative*–that is, from **HOW** to **WHAT**. HOW means saying how, step by step, the work is to be *done*; WHAT just means saying what the work to be done *is*.

So why is this trend A Good Thing? The answer is, of course, that declarative (WHAT) means the system does the work, while procedural (HOW) means the user does it. In a nutshell:

Declarative is better than procedural!

So wouldn't it be nice if we could do *all* of our application development work declaratively? Such has indeed been a goal for many, many years (people have been talking about the possibility of **fully compilable and executable specifications** ever since the 1970s, if not earlier [27]). In other words, wouldn't it be great if we could simply *specify* our applications precisely, and get the system to *compile* those specifications into executable code?

Well, we're getting there. As we'll see.

The advantages are obvious: *Productivity*, of course–the work gets done much more easily and much more quickly; and numerous subsidiary benefits follow, including in particular various kinds of *independence*. One familiar kind is *data* independence, which lets us make changes (for performance reasons, for example) to the way the data is physically stored on the disk, without having to make any corresponding changes in applications that use that data. And there are many other kinds of independence too, some of which we'll be looking at later in this book. The basic advantage in all cases is that they make applications immune to certain kinds of change (in particular, immune to certain kinds of *business* change). And that's a good thing, because–as we all know–the only thing that's constant in life is change.

But what exactly is an application? Obviously, it's the implementation of some business function–for example, "insert an order line item," "delete an order line item," "update the quantity on hand of some part." In general, an application involves three parts or components:

1. Presentation aspects
2. Database aspects
3. Aspects specific to the business function *per se*

Presentation aspects are the ones having to do with the end-user interface–displaying forms to the end user, accepting filled-out forms from the end user, displaying error messages, producing printed output, and so forth. Database aspects are the ones having to do with retrieving and updating database data in response to end-user requests and end-user entries on forms (they're the portions that interact with the database server, also known as the DBMS). Finally, "aspects specific to the business function *per se*" might be thought of as the application proper–they're the ones that specify the actual processing to be carried out in order to implement the business function, or in other words the ones that effectively implement the business's policies and practices.

Now, of these three components, the first two have been largely automated for some considerable time. Application developers no longer write detailed code to paint screens or look for changes in forms on screens–they just invoke builtin *presentation services* to get those tasks done. Likewise, they don't write detailed code to manage data on the disk, they just invoke certain builtin *database services* to get *those* tasks done.

But the third component–the aspects specific to the business function *per se*–those are still mostly done "by hand," meaning that somebody typically still has to write a lot of procedural code. And, of course, that's the problem ... **It's time to automate that third component!** And that's really what "business rules" are all about: *automating the business processing.*

The following back-of-the-envelope analysis might help to give some idea of the magnitude of what we're talking about. A typical database table might need, say, five pages of supporting procedural code; a typical database (not a large one) might contain 100 tables; a good hand-coder might produce a page of code per day. Net: 500 person-days for a hand-coded system. By contrast, simply *specifying* that same system might be a matter of, say, five or six weeks–and if those specifications are executable, then we've effectively reduced the development time by an order of magnitude.

As you can see, automating the business processing, if it could be done, really would be a quantum leap forward. (Sorry about the cliché, but I really do think what we're talking about here deserves the description–much more so, indeed, than certain other "quantum leaps" that have been much discussed in the past.) Writing procedural code is tedious, and time-consuming, and error-prone, and leads to the well-known application backlog problem, as well as many other related problems that we'll get to later. And the "in principle" solution to all of these problems is obvious:

Eliminate the coding!

That is, specify business processing declaratively, via business rules–and get the system to compile those rules into the necessary procedural (and executable) code. And just how we might actually be able to do this is the subject of the next few chapters.

Having set the scene, as it were, let me close this introductory chapter with a quote to support some of the things I've been saying. It's taken from an interview with Val Huber of Versata Inc., formerly known as Vision Software [14].

> Years of experience with information system development have taught us two important lessons–it takes far too long to turn a relatively simple set of requirements into a system that meets user needs, and the cost of converting existing applications to new technologies is prohibitive ... The factor underlying both of these problems is the amount of code it takes to build a system ... If code is the problem, the only possible answer is to eliminate the coding by building systems directly from their specifications. That's what the rule-based approach does.
>
> –Val Huber

As you can see, Huber is effectively suggesting, again, that what we should be aiming for is compilable and executable specifications. Observe in particular that he touches on two separate problems with the old-fashioned way of doing things:

- The time it takes to build applications in the first place;
- The difficulty of migrating existing applications to take advantage of new technologies as they become available (for example, moving a client/server application on to the Web).

I'll come back to both of these problems in Chapter 6.

chapter 2

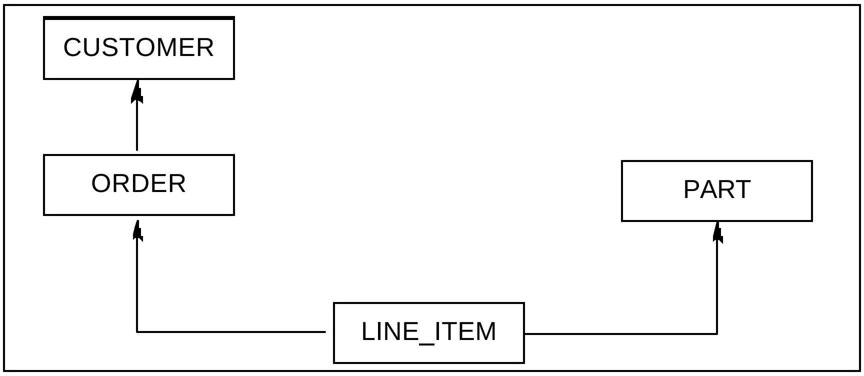
Business Rules Are the Solution!

To repeat from Chapter 1, the problem we're faced with is, in essence, the problem of having to write procedural code. And business rules really are the solution to that problem, because they allow us to specify applications declaratively instead of procedurally.* As I put it in that chapter, business rules allow us to *automate the business processing*.

---

* By the same token, object technology is not the solution, in my very firm opinion. Object languages–Java, C++, and so forth–are still essentially just procedural programming languages (and so someone, somewhere, is still writing a lot of procedural code), and they seem to have no concept of declarative rules of any kind.

Let's take a closer look at what the business rules approach involves. Suppose we have a database involving customers and orders–more precisely, customers, orders, order line items, and parts–that looks like this:



Just to be definite, let's assume this database is an SQL database specifically. Then the boxes in the figure correspond to SQL tables, and the arrows correspond to foreign keys that relate those tables to one another, logically speaking. For example, there's a foreign key from the ORDER table to the CUSTOMER table, corresponding to the fact that every individual order must be placed by some customer.

By the way, if you'd rather think of CUSTOMER and the rest not as SQL tables as such but rather as entity types, well, that's fine; in some ways, in fact, it might be better to talk in such terms, since they're not so specific. Business rules aren't specific to SQL databases! But I'll stick to SQL, for the reason already given.

Observe now that:

- Each customer has many orders (but each order is from just one customer); each order has many line items (but each line item belongs to just one order); each part is involved in many line items (but each line item involves just one part).

- The foreign key relationships can be thought of, in part, as *existence dependencies*: An order can't exist unless the corresponding customer exists, an order line can't exist unless the corresponding order and part both exist.

- Those existence dependencies are business rules! Indeed, foreign key constraints in particular are an important special case of business rules in general, and I'll have quite a lot more to say about them in Part II of this book.

Here then, in outline, are SQL definitions–that is, CREATE TABLE statements–for the four tables in the customers and orders database:

```
CREATE TABLE CUSTOMER
       ( CUST# ... ,
         ADDR  ... ,
         CREDIT_LIMIT ... ,
          ... ,
         PRIMARY KEY ( CUST# ) ) ;

CREATE TABLE ORDER
       ( ORDER#  ... ,
         CUST# ... ,
         PAID ... , --------------- yes or no
         SHIPPED ... , -------------- yes or no
          ... ,
         PRIMARY KEY ( ORDER# ) ,
         FOREIGN KEY ( CUST# ) REFERENCES CUSTOMER ) ;

CREATE TABLE LINE_ITEM
       ( ORDER# ... ,
         LINE# ... ,
         PART# ... ,
         QTY_ORD ... ,
         ORD_PRICE ... , ------ fixed at time of order
          ... ,
         PRIMARY KEY ( ORDER#, LINE# ) ,
         FOREIGN KEY ( ORDER# ) REFERENCES ORDER ,
         FOREIGN KEY ( PART# ) REFERENCES PART ) ;

CREATE TABLE PART
       ( PART# ... ,
         CURRENT_PRICE ... ,
         QTY_ON_HAND ... ,
         REORDER_LEVEL ... ,
          ... ,
         PRIMARY KEY ( PART# ) ) ;
```

Some points to note regarding these definitions:

- The CUSTOMER table includes a CREDIT_LIMIT column, with the obvious semantics.

- The ORDER table includes two yes/no columns, PAID and SHIPPED, that indicate whether the customer has paid for the order and whether the order has been shipped, respectively.

- The LINE_ITEM table includes the part number (PART#), the quantity ordered (QTY_ORD), and the corresponding order price (ORD_PRICE). The order price is locked in at the time the order is placed and doesn't change, even if the current price of the part does subsequently change.

- The PART table includes the current price, the quantity on hand, and the reorder level, all with the obvious semantics.

- Finally, note that the PRIMARY KEY and FOREIGN KEY clauses correspond to business rules (and of course they're stated declaratively). For example, it's a business rule that every customer has a unique customer number; as mentioned earlier, it's also a business rule that every order involves exactly one customer; and similarly for all of the other primary and foreign key declarations.

Now let's consider a typical business function involving this database–"insert line item," say. The way it works goes something like this:

- By clicking on a menu item or something of that nature, the end user asks for a form corresponding to the LINE_ITEM table to be displayed on the screen.

- You can think of that form as a form in the style of QBF (recall that we discussed QBF briefly in the previous chapter): It will include among other things fields corresponding to the customer number, the order number, the part number, and the quantity ordered.

- The end user fills in those fields appropriately–that is, he or she provides the necessary information regarding the new order line–and clicks on "enter" or "save" or something of that kind.

The "insert line item" application is now invoked and carries out the following tasks (among many others, of course):

A. It checks the customer's credit limit.

B. It computes the order total.

C. It determines whether the part needs to be reordered.

A., B., and C. here are business requirements that must be met in order to carry out the overall business function. Incidentally, there's an important point here that I'll come back to in a little while (when I discuss *reuse*): Those very same business requirements might also need to be met as part of certain other business functions–for example, "change line item" or "delete line item." But let's concentrate on the "insert line item" function for the time being.

For each of these three business requirements, then, the application developer will have to specify a corresponding set of business rules. In the case of the "check credit limit" requirement, for example, the rules might look something like this (of course, I'm simplifying the syntax considerably, for obvious reasons):*

* It's worth mentioning that–as will quickly become clear–the process of developing these rules precisely mirrors the "analysis interview" process: The analyst gets the business user (a) to state the business objectives and (b) to provide definitions of terms that get introduced in step (a).

```
1. If TOTAL_OWED > CREDIT_LIMIT, reject
```

The meaning of this rule, obviously enough, is that the new line item must be rejected if it has the effect of carrying the total owed by this customer over the customer's credit limit. But what do we mean by "the total owed"? Clearly, we need another rule:

```
2. TOTAL_OWED = Sum ( ORDER_TOTAL where not PAID )
```

Note that there isn't any "total owed" column in the CUSTOMER table, so the total needs to be computed as indicated.

Observe now that this second rule refers to the order total. So this rule leads us directly into the second business requirement, "compute order total," for which the rules might look like this:

```
3. ORDER_TOTAL = Sum ( LINE_ITEM_AMOUNT )

4. LINE_ITEM_AMOUNT = QTY_ORD * ORD_PRICE
```

QTY_ORD and ORD_PRICE are both specified as part of the line item (the line item to be inserted, in the case of the "insert line item" business function), so LINE_ITEM_AMOUNT can be computed directly.

Finally, here's the rule for the third business requirement, "determine whether reorder is required":

5. If QTY_ON_HAND - QTY_ORD < REORDER_LEVEL, reorder

"Reorder" here can be thought of as the name of another application, part of the same overall integrated application system (or what we used to call the *application suite*). Alternatively–and this is a very important point–"reorder" might mean "send an email message to some external agency." So we're not just talking about calling subroutines!–and we're not just talking about applications in the classical sense. I'll have a little more to say on this point in Chapter 8.

As you can see, then, these rules are fairly declarative (nonprocedural). **But they can be compiled into procedural code.** In other words, the rules are *executable* (loosely speaking). So we've specified the application in a purely declarative way–we haven't explicitly written any of the usual procedural code at all–and yet we've still wound up with running code: an application that can be executed on the machine.

By the way, the procedural code produced by the compiler isn't just executable code–it is (or should be) *optimized* code as well. That is, the rules compiler is (or should be), specifically, an optimizing compiler. I'll have more to say on this particular point in several subsequent chapters.

That's the end of the example for now (though we'll be coming back to it at several points in the next few chapters). To repeat, we've just built an application without writing any procedural code! And there are many immediately obvious advantages that accrue from this way of doing things. Here are some of them:

- First of all, of course, the declarative rules replace many pages of hand-written procedural code. Each of those rules could easily correspond to a couple of hundred lines of 3GL code! This is the source of the productivity benefit, of course.

- Next, the rules are applied and enforced–"fired," to use the jargon–*on all relevant updates*. I touched on this point before, when I observed that the very same business requirements might need to be met as part of several different business functions. To be more specific, although the rules we looked at in the example were specified as part of the process of building the application called "insert line item," *they're relevant to other* *applications too*. For example, the application called "delete line item" should clearly also cause the rule "compute order total" to fire, *and so it does*.

To put the point more simply: The application developer doesn't specify *when* the rules are to fire, or what events have to occur in order to trigger them. (We don't want to have to specify those triggering events, for all kinds of reasons. One obvious and important reason is that we might get them wrong.) Rather, we simply say what the rules *are*, and the system itself–the "rule engine," as it's sometimes called–figures out when they should fire. Productivity!

- It follows from the previous bullet item that there's no need for the application developer to get into the business of writing that complicated "on event" hand-code that's required with certain 4GLs. (It's relevant to mention too that those 4GLs were often still fairly procedural–not to mention proprietary–even when they were "event-driven," as it were.) And there's another point here as well: Not only is that "on event" hand-code difficult to write, it's also difficult to debug and maintain. What's more, the events in question tend to be, specifically, *database* events–for example, "when this record is updated"–rather than *business* events like "when an order line is entered."

  Note: Actually the foregoing paragraph is slightly overstated. Sometimes we do effectively have to define "on event" rules after all. Rule 5 above, the reorder rule, might be regarded as a case in point; in that example, the triggering event is basically "when the quantity on hand falls below the reorder level" (though the rest of the rule can hardly be described as "complicated hand-code"!). Be that as it may, I propose to ignore such cases from this point forward, until we get to the discussion of *stimulus/response* rules in Chapter 5.

- It also follows from the previous bullet item that rules are automatically *shared* across applications and *reused* across applications. It's rather like the situation with the database itself: The data in the database is also shared and reused across applications, as we all know. And the advantages of such sharing and reuse for rules are analogous to those for sharing and reuse for the data itself. In effect, just as the relational model originally allowed us to integrate, share, and reuse *data*, so business rule technology allows us to integrate, share, and reuse *applications* (or pieces of applications, anyway).

- Another advantage of declarative rules is what might be called the "single-level store" advantage. To be more specific, the rules make no mention of any kind of artificial boundary between the database and main memory; data is data, wherever it resides, and there's no need to move data out of the database and into main memory in order to process it. (No need so far as the user's concerned, that is. Of course the data does have to be moved under the covers. But why should the user care? Let the system do it!)

- Finally, note that the rules can be stated *in any sequence.* To say it another way, we have *ordering independence* for the rules. Business rule technology allows us to get away from the tyranny of the von Neumann architecture, which forced us to think everything out in a highly procedural manner, one step at a time. As Gary Morgenthaler has said [20]:

  > "[With the business rules approach,] programmers ... are freed from the burdensome task of having to redefine their business problems in the deadening pursuit of stepwise instructions for the program counter."

  Down with the program counter!

  Of course, the procedural code that's compiled from the rules does have to pay attention to the program counter (the rules do have to *fire* in some well-defined sequence). So how is it done? How does the rule engine figure out that sequence? Basically, it does so by means of what's called a *dependency graph*. In the example:

  1. In order to check the credit limit, the system needs to know the total owed;

  2. In order to compute the total owed, it needs to know the line item amounts;

  3. In order to compute the line item amounts, it needs to know the line item quantity and price. But it does–these values are given by columns in the LINE_ITEM table (or are provided by the end user, in the case of a new line item that's currently being inserted).

  So the dependency graph shows that 1. depends on 2. and 2. depends on 3.–so (obviously enough) the system fires the rules in the sequence 3., then 2., then 1.

  By the way, note the reliance in the foregoing on the foreign key constraints (which are rules too, of course, as we already know). For example, the rule

  ```
  TOTAL_OWED = Sum ( ORDER_TOTAL where not PAID )
  ```

  implicitly makes use of the foreign key from ORDER to CUSTOMER (the order totals to be summed for a given customer are precisely those for orders that have a foreign key relationship to the customer in question). Note: Foreign keys can also be referenced explicitly, of course, where such explicit reference is necessary.

I'll have more to say regarding the advantages (and potential *dis*advantages) of the business rules approach in Chapters 6 and 7. For now, I'd like to close this chapter with a brief remark on the question of terminology. The fact is, the term "rules" might be convenient, but it's hardly very precise, or even very descriptive. After all, we already deal with all kinds of rules in the database world, or the application world, or both. Here are a few examples:

- Security rules
- Integrity rules
- Codd's "12 relational rules"
- Foreign key rules
- BNF production rules
- Type inference rules
- Armstrong's functional dependency rules
- Expression transformation rules
- Logical inference rules

I'm sure you can provide many more examples of your own.

What's more, the same goes for the term "*business* rules" as well; that term also is neither very precise nor very descriptive. Note in particular that not all enterprises are businesses! But it doesn't seem to be easy to come up with any better terms–at least, not ones that are so succinct–so I'll go with the flow in this book and stick with the term "business rules" (which I'll abbreviate to just "rules," usually). However, the key point is really **declarativeness;** the important thing is really that we're talking about a *declarative* technology, not a procedural one.*

* However, to quote Paul Irvine, who reviewed a preliminary draft of this book: "The term *business rules* gives the concept a very marketable tag ... *declarative programming* just won't sell!" A sad comment but undoubtedly true.

So what exactly is a business rule? As this book uses the term, it can be any of the following (but be warned that not everybody uses the term in exactly the same way):

1. A presentation rule
2. A database rule
3. An application rule

In other words, rules apply to all three aspects of applications as identified in Chapter 1.
In the next couple of chapters, we'll take a closer look at each of these three kinds of rules.