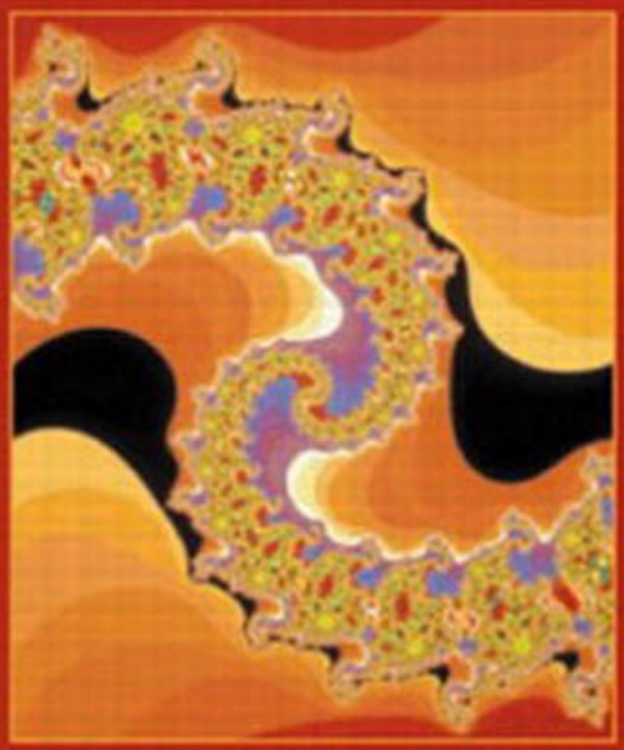
Business Rules and Information Systems

Aligning IT with Business Goals



Tony Morgan

Library of Congress Control Number: 2002101260

Text printed on recycled and acid-free paper.

ISBN 0-201-74391-4

4 5 6 7 8 9 DOC 10 09 08

4th Printing January 2008

Contents

Preface

Why this book

It seems that every week brings a new story about a software project or a system that's failed in some way. Quite often, the failures are so big that they make it to the national press. Most of the information given is anecdotal and circumstantial, but even a nontechnical observer might suspect that something is seriously wrong with the way we develop software systems.

My own view is that the core of the problem lies in the casual way we treat "the requirements": the statements that tell us what an information system is supposed to do. These statements are typically captured only in a rudimentary way, are poorly structured, and are linked to the software only by ideas in the heads of analysts and developers and so aren't open to examination or verification.

If we can return "the requirements" to a more prominent role in the process and use them to drive the subsequent development stages, we have the potential for a drastic reduction in the number of errors. Adding automation to this process can further reduce the opportunity for error and, as a bonus, also give big reductions in time and cost. We haven't been able to do this in the past, because there's been no clear strategy that we could adopt to drive things forward.

The idea of business rules, rooted in a business model, might provide part of the answer. But practical information on the topic is unexpectedly sparse, even though most of the basic ideas have been around for some time. Ideally, this book will help plug the gap.

The goals of this book

I wrote the book to pull together a load of separate strands and to show how they fit together to provide a coherent foundation for building information systems. In truth, very little in the book is completely new, and maybe that's a good thing. What we need is not so much new technology as a renewed focus on what's important.

The intention is not to convince you to use rules. You're already using them. In fact, if you're in a large organization, you probably have thousands of them. They guide the way your organization works, they make sure that you comply with legal and other regulations, and they are a source of competitive advantage or a barrier to success. But in most organizations, rules lead a shadowy existence. They may be enforced by your information systems, so that you're always directed toward the goals the business has adopted. But then again, the rules may not be so enforced.

You need to get an adequate level of control over your environment. I cannot know about your specific circumstances, but maybe the material in this book will help you to come up with some actions that put you in a better position to be a leader, not just a survivor, in your industry.

Although the information here is probably enough for you to develop your own complete set of tools and processes for building rule-centered systems, it's not really what I would recommend. Generally speaking, you should prefer a properly supported commercial product—if one exists—to a home-brewed tool. But all products have their little foibles, and before you fall into the arms of a particular vendor, you need to understand what you're gaining and what you're giving up. The information in this book should help you to ask the right questions before you make a commitment. You can also use the material in this book to help you to define any local "glue" that may be required to make your tool set stick together properly.

This book contains no information about particular commercial products or vendors. This comparatively new area for tool support is an immature technology, and is undergoing rapid change. Any descriptions of current tools would be out of date in a few months or even weeks. The best sources of information are the suppliers' Web pages, where you'll usually find product descriptions, white papers, and other supporting materials. Just search for terms like "business rule," pull out the sites that are on topic from the list returned, and build up your own set of favorites. You should also check out the Addison-Wesley Web site at www.aw.com/cseng/, where you'll find more information relating to this book.

Who should read this book

This book is aimed primarily at professionals working in the field of information technology (IT). If you have any involvement in the definition, creation, or management of an information system, you should be able to gain something of value from this book.

Analysts, responsible for capturing requirements and for specifying information systems, can find out more about producing logically complete definitions of the needs of the business. This includes understanding business models and their

various constituents, knowing how to locate business rules, and determining how to express them in a form that maintains their true value.

Designers and developers, with responsibility for the implementation and testing of systems, can find practical examples of how business logic, expressed in the form of rules, can be conserved and taken forward into operational software. This also provides for two-way traceability between the worlds of specification and implementation.

Managers and strategists are obliged to take a higher-level view of the whole process. What they will find here are practical steps to help them to manage the intellectual property represented in a system, along with ideas for improving the development process in order to deliver information systems faster, cheaper, and, above all, to a level of quality that can far exceed current ad hoc methods.

How to use this book

Please don't treat this book as a set of edicts about what you should or should not do. It's meant to be a source of ideas that you can meld into your own approach to the needs of information systems in the twenty-first century.

The thing that resonates with me most strongly after engagements in a large number of IT environments is that they are different! Of course, there are similarities from place to place, but no one wants to be just the same as everyone else in their market sector. In fact, you can't really afford to be a "me too" player, who at best expects to survive, not to be a winner.

Using IT effectively requires you to balance out two different things.

- You need to be realistic about what technology can provide but also be prepared to take up new capabilities as they arise.
- You have to look for ways in which you can differentiate your operation by doing it faster, cheaper, or to a level of quality that the competition can't match.

The material in this book is aimed at providing you with the information you need to make crucial decisions in this area. I can't tell you how to run your business, but I can provide pointers to ideas that you may be able to use to lead your industry in the application of information technology.

The content falls into four main parts. Part I—A New Approach to Information Systems—sets the scene by suggesting how we could begin to approach information systems in a different way. Chapter 1 paints a picture of an alternative future that uses structured descriptions of a business to drive system development. Chapter 2 fills in some of the background on what we mean by structuring and managing knowledge about a business. This chapter introduces business models

and the role they can play in meeting the demands of new business directions, such as e-commerce.

Although business rules have a particularly important part to play here, they are probably the least well documented of all the business model elements. Part II—Capturing Business Rules—therefore delves into rules in greater depth and provides some fairly detailed information that should help you to set up a sound framework for delivering logical business descriptions. Chapter 3 explains how to define business rules in a systematic way. Chapter 4 discusses how to identify business rules and pull them into a managed environment. Chapter 5 shows how the business logic that underlies the rules can be validated, providing assurance that the intentions of the business have been captured accurately so that later stages can proceed with confidence.

In Part III—Implementing Business Rules—we take a look at the other end of the process and consider realistic mechanisms for the implementation of information systems and business rules in particular. Chapter 6 reviews the kinds of technical architectures that dominate in most organizations and shows where rules can fit into the kinds of structure that are likely to be available. Chapter 7 goes into more detail on the various ways that business rules can be realized using readily available technology. Chapter 8 discusses ways of managing rules and models and the part they play in information system development.

Finally, Part IV—The Role of Business Rules—rounds things out by summarizing the current state of play. Chapter 9 shows how business rules build on longstanding ideas about structuring descriptions of interactions between people and between people and machines and points to some directions that this may take in the future. Chapter 10 gives a summary of the main characteristics of business rules and the value they can provide.

The Appendix summarizes the key elements of logic that need to be understood by anyone working with business rules. If you're entirely comfortable with the ideas of formal logic, you can skip this material, but you may want to dip into it if you feel the need for a refresher.

Most of all, I would be happy if this book encourages you to think about information systems in a different way. Let's focus on producing a logically complete description of what we want and let the machines take care of the details.

3

Defining Business Rules

3.1 Rule statements

In this chapter, we concentrate on what a business rule looks like. This begs the question of how we find rules in the first place, but that's the subject of the next chapter. On balance, it's probably better to start with a description of what to look for rather than how to find it.

3.1.1 Business rule characteristics

Broadly speaking, business rules are constraints: They define conditions that must hold true in specified situations. Sometimes called invariants, business rules are not descriptions of a process or processing. Rather, they define the conditions under which a process is carried out or the new conditions that will exist after a process has been completed. (The word *process* is used here in the general sense.)

Putting this another way, business rules define *what* must be the case rather than *how* it comes to be. A set of rules that define pre- and postconditions can act as a specification for a process, without constraining the mechanisms through which the preconditions are transformed into the postconditions.

Business rule statements in the business model define the desired logic of the business. They describe a state of affairs that the business wants to exist—in fact, what the business demands. If they were to be expressed as Boolean functions, business rules would always return a value of true; otherwise, they would be pretty useless as a definition of the desired logic. From a logical business rule perspective, there are no exceptions; there are only rules. A supposed exception to a business rule is simply another business rule.

You should not confuse this with what happens after the rule has been implemented. During processing, conditions may exist that would result in false if the business rule were to be evaluated. The whole point of the business rule approach

is that it identifies the conditions that need to be detected and the corrective action that's required to restore the system to a state in which "true" always prevails. Typically, these are inside transactions, which we'll consider in more detail in a later chapter. Once a transaction has reached a stable state—has completed or aborted—it must not leave the system in a condition that contravenes any business rule.

The job of the business analyst is to specify a series of clear statements about the logic underlying a business. The emphasis on clarity is crucial. The business rule statements must be in a form that the business owner can immediately accept as valid or reject as invalid. If you are an analyst, this is your job. A retreat into "technospeak" is an admission of failure. It's no good complaining that the businesspeople don't understand first-order predicate calculus or whatever you may think is particularly cool from a technology perspective. The fact that they already operate the business without using anything more than simple natural-language statements is a strong proof that esoteric notations, technical languages, and so on, are not essential.

More than two thousand years of experience have shown that it's perfectly possible to make clear, logical statements using nothing more than ordinary words: in English, French, Chinese, German, Japanese, Spanish. The language doesn't matter as long as the logic is clear. To be sure that what you're saying is really sensible, it's useful to break a description of a complex system down into a set of small, manageable units, like logical propositions. If you're not sure what this means, the appendix gives an overview of how to form and to manipulate logical statements.

On its own, one business rule statement may not look very impressive. It may say something about the business that seems trite, obvious, and hardly worth the trouble. Don't worry; this is a good thing. If it seems obvious, you've cracked the most difficult problem: making the business logic understandable.

The real power comes from two directions:

- The ability to make business-level statements that can be translated in a fairly direct way to become part of an operational system
- The combined effect of relatively large numbers of simple statements, so that together, the whole has an impact that's greater than the sum of the individual parts.

The starting point for all this is nothing more than a series of simple statements about the business. There's nothing technical or difficult to understand here. What we're trying to do is so simple, it seems patently obvious. The strange thing is that so many organizations become diverted by other concerns, especially political ones, that they forget the obvious and get diverted onto side issues.

Let's start with a simple list of the characteristics that we would like to see in business rule statements. These universal characteristics apply across any language, tool, application domain, or any other sort of division. We're looking for business rule statements that are

- Atomic: can't be broken down any further without losing information
- Unambiguous: have only one, obvious, interpretation
- Compact: typically, a single short sentence
- Consistent: together, they provide a unified and coherent description
- Compatible: use the same terms as the rest of the business model

In the following sections of this chapter, we'll see how to put together statements that fit with this agenda.

3.1.2 Business aspects

At a high level, business rules could be classified under one or more concerns, such as the following:

- Reducing risks to the business or minimizing their impact
- Improving customer service
- Making the most efficient use of corporate resources
- Controlling or managing the flow of work

However, knowing this does not necessarily help with identifying the rules in the first place. At a finer level of detail, rules are commonly associated with various aspects of the business; Table 3-1 summarizes a few of the most common ones. We'll look at the process of rule discovery in more detail in the next chapter.

Table 3-1 Some uses of rules

Aspect	Examples
Consistency of information	Dates, modes of address, corporate styles, and so on that should be treated in the same way throughout the organization
Entity relationships	Relationships that must be enforced between entities relevant to the business, perhaps under certain conditions; for example, a Party must have at least one Contact Point
Identification of situations	Recognition of common business situations, allowing standardized, predictable, and well-managed responses to be made
Data integrity	Default values, algorithms for computation, checks on correctness of values, and other means of ensuring the correctness of data

3.1.3 What should a rule say?

Business rules should be concerned only with the conditions that must apply in a defined state. If you're writing a rule statement, you should resist the temptation to

add extra information that goes outside this boundary. The extra information should be covered elsewhere; by duplicating it in a rule, you are only increasing the potential for error.

In particular, a business rule should define *what* should be the case and should not prescribe

- *Who* invokes the rule. (This is usually described in a use case or a process description.)
- *When* the rule is executed. (This is usually described in a business event, a use case, or a process description.)
- *Where* the rule executes. (This will be defined in design.)
- *How* the rule is to be implemented. (This will be defined in design.)

This point sometimes causes confusion, so it's worth being extra clear about it. We're not saying that rules should be prohibited from mentioning people, places, times, and so on. It's more a question of describing a state of affairs that should exist.

For instance, consider a rule that starts

> During the final quarter, all managers based in Europe must . . .

This does not specify *who* invokes the rule. It's not necessarily any of the managers referred to. It could be their secretaries, their bosses, an automation system, any combination of these, or different ones at different times.

It does not specify *when* the rule is executed. The rule is applicable only in a given period, but, depending on the choice of implementation, it could be checked at the end of each working day, once at the end of the quarter, with every database transaction, or any other time the business decides.

Nor does it specify *where* the rule is executed. The rule defines what happens in a specific geographical area, but it could be executed at the head office, at each manager's location, in a remote automation system, or wherever is most convenient for the business.

A less obvious case is the question of *why* the rule is applied. Let's pull apart two separate issues:

- *Why* are we bothering with this rule in the first place?
- If processing fails because of a particular rule, can we tell *why* this is happening?

The answer to the first question is a fairly simple consequence of the system scope. The rule is relevant if it's something we need to say to pin down an aspect of the system. If we need it, we say it; if not, we don't. We'll have more to say about defining and recording the source of a rule in later chapters.

The second question is a bit more interesting and reflects a common problem with today's systems. If something goes wrong, how can we tell what's happening? A description of the problem at a technical level might only add to the confu-

sion. For example, an error message, such as "interface method addParty failed" is not very helpful to a businessperson. Such a failure, which, of course, will require rectification before processing can continue, is best explained by the business rule that identified the problem.

For instance, let's say we have this business rule:

```
R301    A Party must be related to at least one PartyRole.
```

If we tried to create an instance of a Party without defining the role that the Party plays, what should the system do? Clearly, it should not accept the existence of the Party, given that particular business rule. What would be the best explanation of the failure? It is the business rule itself; read R301 again. This is a point to take on board when creating business rules: Thinking through the immediate business pressures, how could we answer the *why* question?

3.1.4 Levels of expression

So far, we've glibly talked about rule statements as one simple kind of thing. In fact, it's possible to imagine at least three levels of rule expression. All have a structure but occupy different points along the trade-off between accessibility of business meaning and desirable automation properties. The following examples illustrate what a rule might look like at each of the levels. (These aren't meant to correspond to any particular language or notation standard.)

1. *Informal.* This provides colloquial natural-language statements within a limited range of patterns. For example:

   ```
   A credit account customer must be at least 18 years old
   ```

2. *Technical.* This combines structured data references, operators, and constrained natural language. For example:

   ```
   CreditAccount
   self.customer.age >= 18
   ```

3. *Formal.* This provides statements conforming to a more closely defined syntax with particular mathematical properties. For example:

   ```
   {X , Y, (customer X) (creditAccount Y) (holder X Y) }
   ==> (ge (age X) 18)
   ```

The quest for greater structure points to the most formal option as being the most desirable, but most people at the business level would be far happier with the more colloquial informal option. One way of resolving this is through decent

tool support. Figure 3-1 shows a low-technology way of creating and using rule statements.

The analyst creates rules at an informal level, treating the rules as pieces of text. The text does have the advantage of keeping the rules easy to read, but all the control over rule structure, consistency, and so on, has to come from the discipline of the analyst. The translation into formal structures, leading ultimately to one or more implementations of the rule, is similarly a human activity, with consequent opportunities for the introduction of errors. This is roughly the level of support for business rules provided by the current generation of tools.

What's needed is a way of creating the more formal rule structure while retaining the ease of use of the colloquial statements. Figure 3-2 sketches how this might be achieved. The job of the analyst is now to manipulate a set of predefined structural units. The various structural forms can be used as a basis for generating the equivalent textual representations; they are not edited directly. Although the appearance to the analyst and to the business is still in the form of text, all the control over the structure is now within the system. Given a reliable structure to work from, it's now also possible to think about generating code from the structure.

We can take this even further. Figure 3-2 shows the analyst as an intermediary between the business owner and the rule definition, but human interpretation allows the introduction of errors. The ultimate goal has to be an arrangement closer to Figure 3-3, with the business owner having direct control over the rule definitions. Because present tools are not sufficiently mature, however, this is not a practical option today, but it's definitely the direction we should be taking.

In the rest of this chapter, we'll talk mainly about text-based rule statements, of the kind described as informal, but with an eye to the improved tool support we can expect from future generations of analysis and design tools.

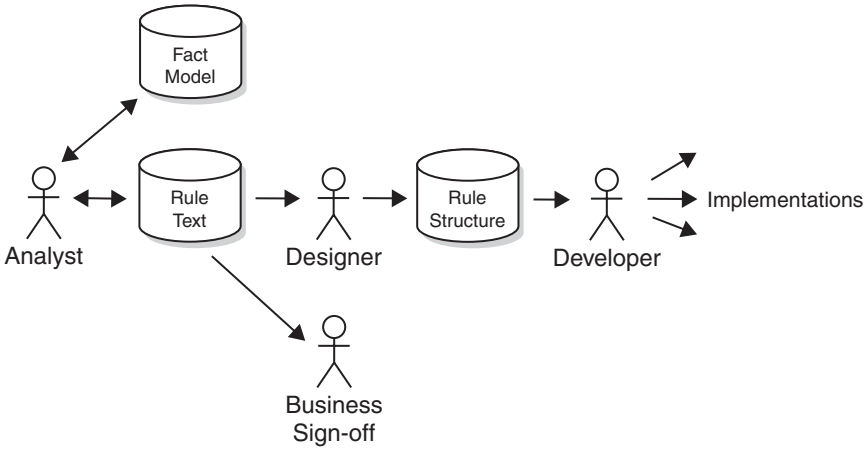


Figure 3-1 Low-technology rule definition

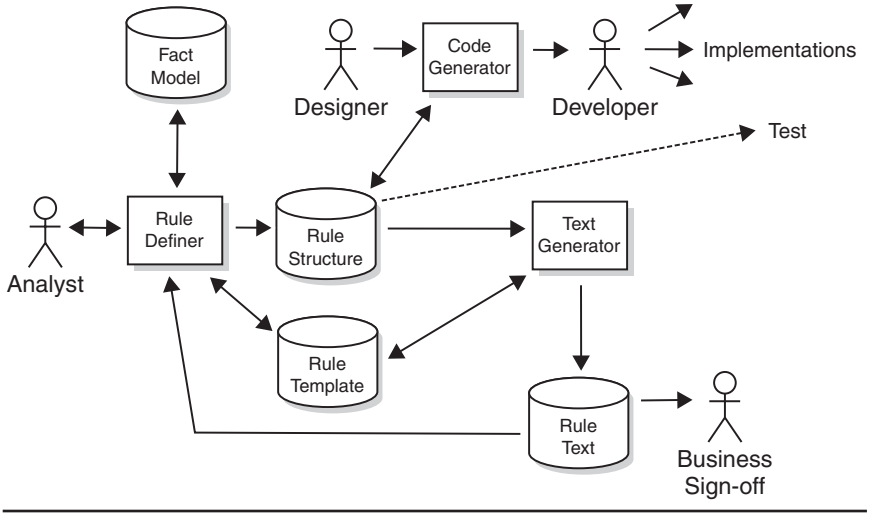


Figure 3-2 Controlled rule definition

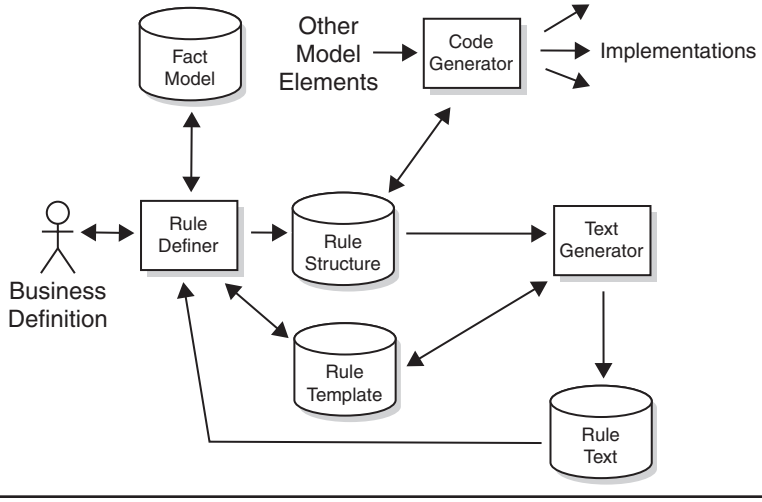


Figure 3-3 Rule definition: long-term objective

3.1.5 OCL

Although we've discussed a number of the notations used in UML, we haven't yet mentioned one of its less well-known features: the Object Constraint Language (OCL), which was added in UML version 1.1. See Warmer and Kleppe 1999 for

more details. OCL is a language for expressing constraints on object-oriented models. It supplements the diagramming aspects of UML by allowing the modeler to say things that the diagrams don't support well or clearing up ambiguities that often seem to arise when UML class diagrams try to show complex relationships. In terms of the three levels described earlier, OCL fits somewhere around the technical level but perhaps somewhat on the formal side.

For example, suppose that we have a Person class with an attribute socialSecurityNumber. We want to have a rule that a person's social security number must be unique. That wording is more or less how we would express it in a rule at the informal level. Here's how it looks at a technical/formal level in OCL:

```
Person
Person.allInstances -> forAll (p1, p2 |
     p1 <> p2 implies   p1.socialSecurityNumber <>
                        p2.socialSecurityNumber)
```

If you read this carefully, you can pretty much tell what it's saying, even if you aren't familiar with the OCL syntax. For the average analyst, creating it from scratch might be a different matter, though. And as far as getting the business owner to sign off on something like this, forget it!

To date, OCL's main claim to fame has been its role in cleaning up the UML specification, but there's very little evidence of its achieving widespread adoption in practical information systems, which is a pity. The apparent lack of enthusiasm undoubtedly results from the daunting appearance the syntax presents to the business owner and also to most analysts. OCL is really usable only by a technical specialist, which rules it out as a candidate for the "visible" rule format that we want to present to the business owner in Figure 3-3 and, probably, also to the analyst in Figure 3-2.

Let's not write OCL off, though. It could have a very useful part to play behind the scenes, as a more precise representation that an informal rule might be translated into. For now, we'll put it to one side because we need to focus on the front-end business statements.

3.2 Forming rule statements

The most convenient way of creating rule statements is to select an appropriate pattern selected from a short list of available patterns. Besides saving valuable analysis time, this helps to enforce some consistency of approach. In this section, we discuss a vanilla set of patterns that should be applicable across a range of types of applications and industries.

The patterns we cover are in the form of *statements*, and they're all more or less elaborate versions of one basic form:

```
<subject> must <constraint>
```

This is in tune with the idea that a rule is always true: The rule makes a statement about what must or must not be the case in the operational system. Both the `<subject>` and the `<constraint>` of the rule can get to be fairly complex, as shown in some of the examples that follow later. A range of patterns allows us to have reasonably natural-sounding business sentences without introducing all the difficulties associated with full natural-language processing.

In addition to conforming to an appropriate pattern, the rule also has to make reference to other model elements, principally business objects and their attributes. This is done through a fact model, which we'll look at in more detail shortly. Figure 3-4 shows the position of rule statements in this scheme of things.

There's no standard for how to make rule statements, although it's possible to make some recommendations, based on experience. You should regard the following set of patterns as a suggestion to start from rather than a mandatory requirement. One reason you might want to use a different set of patterns is to fine-tune the wording for the domain you're working in. These patterns might reflect the way things are done in your industry or the kinds of problems that your automation system is intended to deal with. What's important is not the patterns themselves but the adoption of this style of rule definition. We'll see an example in a case study later in this chapter.

The following patterns have a reasonably neutral flavor: They don't assume any particular kind of application. The only presupposition is the existence of a fact model to provide a context for the subject and the constraint.

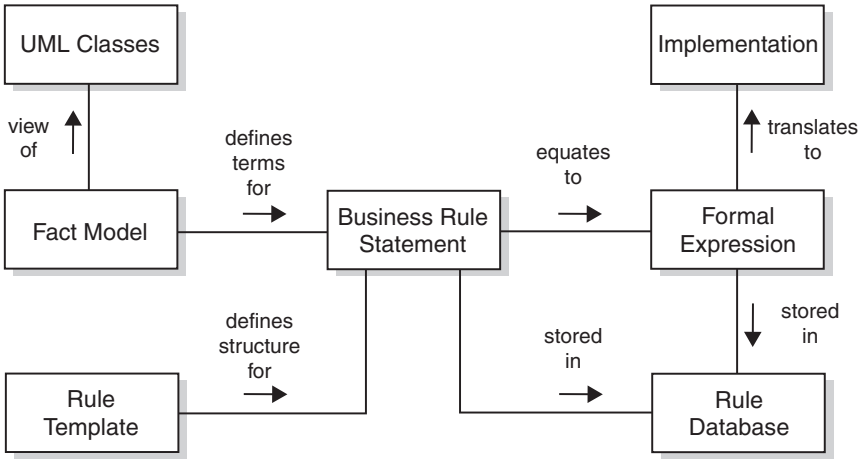


Figure 3-4 Rule statements and their relationships

3.2.1 Pattern conventions

The conventions used in the patterns are as follows.

- Parentheses ( ) enclose a group of terms
- Brackets [ ] enclose optional terms
- Vertical bars | separate alternative terms
- Angle brackets < > enclose special terms, as defined in Table 3-2.

Note that the symbols do not appear in the rule statement but only in the pattern definition.

Table 3-2 Variable pattern elements

Element	Meaning
<det>	The determiner for the subject; from the following, the one that makes the best business sense in the statement: A An The Each Every (or nothing).
<subject>	A recognizable business entity, such as a business object visible in the fact model, a role name, or a property of an object. The entity may be qualified by other descriptive elements, such as its existence in a particular state, in order to specify the applicability of the rule with enough precision.
<characteristic>	The business behavior that must take place or a relationship that must be enforced.
<fact>	A relationship between terms identifiable in the fact model, together with defined constants. The relationship may be qualified by other descriptive elements in order to specify the applicability of the rule precisely.
<fact-list>	A list of <fact> items.
<m>, <n>	Numeric parameters.
<result>	Any value, not necessarily numeric, that has some business meaning. The result is often, but does not have to be, the value of an attribute of a business object.
<algorithm>	A definition of the technique to be used to derive the value of a result; normally expressed using combinations of variable terms identifiable in the fact model together with available constants.
<classification>	A definition of a term in the fact model. This typically defines either the value of an attribute, perhaps called “state” or something similar, or a subset of the objects in an existing class.
<enum-list>	A list of enumerated values. An *open* enumeration indicates that the list may be modified in the light of future requirements; for example, a list of status values for an object as currently known. A *closed* enumeration indicates that changes to the list are not anticipated; for example, days of the week. The distinction is helpful in later implementation.

3.2.2 Rule patterns

Pattern 1: Basic constraint This pattern, the most common business rule pattern, establishes a constraint on the subject of the rule. Two equally valid variants are provided; choose the option that gives the most natural business style to the rule statement when you read it back. The optional word "should" in this pattern and the next one makes an easier-sounding expression in some circumstances. It does not make the rule optional in any way. If you think that this may cause confusion in your organization, don't allow its use.

<det> <subject> (must | should) [not] <characteristic> [(if | unless) <fact>].

<det> <subject> may (<characteristic> only if <fact>) | (not <characteristic>).

Examples:

R302 An urgent order must not be accepted if the order value is less than $30.

R303 An account may be closed only if the current balance is zero.

Pattern 2: List constraint This pattern also constrains the subject, but the constraining characteristic(s) is (are) one or more items taken from a list. Again, two variants are provided, so you can choose the one that's the best fit to the particular situation.

<det> <subject> (must | should) [not] <characteristic> (if | unless) at least <m> [and not more than <n>] of the following is true: <fact-list>.

<det> <subject> (may <characteristic> only if) | (may not <characteristic> if) at least <m> [and not more than <n>] of the following is true: <fact-list>.

Examples:

R304 A next-day order must not be accepted if at least one of the following is true:
—Order is received after 15:00, Monday through Friday,
—Order is received anytime on Saturday or Sunday,
—Order is received between 15 December and 5 January,
—Postal inhibition is in place.

R305 A customer may be raised from Bronze to Silver status only if at least one of the following is true:
—The customer's account balance has been positive for the past 12 months,
—The customer's average account balance over the past 12 months exceeds $500.

Pattern 3: Classification This pattern establishes a definition for a term in the fact model. Such terms are likely to have only a short-term, temporary usage. If the distinction is permanent, it may be better to reflect it in the fact model. The variants are just choices for readability.

<det> <subject> is [not] defined as <classification>
[(if | unless) <fact>].

<det> <subject> must [not] be considered as <classification>
[(if | unless) <fact>].

Examples:

R306 An order is defined as urgent if delivery is required in less than three hours.

R307 A customer without credit clearance must be considered as cash-only.

Pattern 4: Computation This pattern establishes a relationship between terms in the fact model sufficient to allow the computation or the establishment of a value. Note that this is similar to the Classification pattern. Using "is defined as" instead of the imperative "must be computed as" leaves the computation implicit in the relationship. This is to avoid an overprocedural style and because it may be possible to compute various values based on the relationship. For instance, in the following example R308 it would be equally valid to calculate "total item value" from "total sale value" and "sales tax". Two variants are provided; the second is more likely to be favored for mathematical expressions.

<det> <result> is defined as <algorithm>.
<det> <result> = <algorithm>.

Examples:

R308 Total sale value is defined as total item value plus sales tax.

R309 Pi = 4 * arctan(1).

Pattern 5: Enumeration This pattern establishes the range of values that can legitimately be taken by a term in the fact model.

```
<det> <result> must be chosen from the following
[ open | closed ] enumeration: <enum-list>.
```

Example:

```
R310    Customer standing must be chosen from the following
        closed enumeration:
        —Gold,
        —Silver,
        —Bronze.
```

3.2.3 Rule sets

For complex cases, it may be useful to group rules into rule sets, or one "master" rule with a number of subsidiary rules. Splitting up a complex problem into smaller parts greatly helps the initial rule definition and is vital for long-term maintainability. An extended example of a rule set is given in the next chapter.

3.2.4 Static models versus rule statements

Some situations require you to exercise judgment about the best way to capture the business logic. The main area for trade-off is between business rules and the fact model. A good example of this is cardinality (also known as multiplicity). Simple cardinality constraints on associations can sometimes be expressed either as a rule or directly in the static model without using a rule statement. A fragment of a fact model with a typical simple relationship is shown in Figure 3-5.

Let's say that we wanted to exercise some control over this and insist that if we hold any information about the existence of a Party, we must also identify the

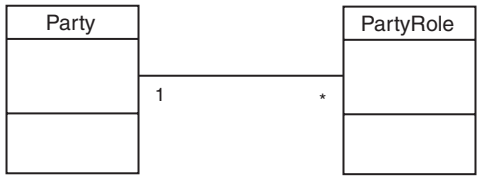


Figure 3-5 Relationship with weak constraint

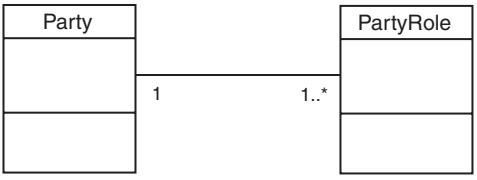


Figure 3-6 Relationship constrained by cardinality on class model

role that Party plays, such as customer or supplier. One way of doing this would be with a business rule, such as:

R311 A Party must be related to at least one PartyRole.

Alternatively, instead of using the business rule, the constraint could be shown using cardinality markings on the class model, as shown in Figure 3-6. In this case, no rule statement would be needed.

However, in general, you should be very wary about introducing additional classes or associations into the fact model purely to express constraints. The logic will be clearer, and more maintainable in the long term, if it's captured in the form of rule statements. We'll return to this later in the book when we get to rule realizations.

A useful guideline—if you can convince your local UML zealots—is to limit the class diagram to showing just one (1) or many (*) on the association ends and stating any elaborations on this as explicit business rules. That way, it's always clear where you should look for any constraints.

3.3 References to facts

3.3.1 Terms and rules

The rule statements refer to various terms—used as the definition of <subject> and <constraint>—that are visible in a supporting fact model. What ties a rule down to a particular situation is the explicit reference to something that's visible in the fact model.

Business rules build on a known set of facts described in a model. Conceptually, this *fact model* shows business objects, their relationships, and their attributes. In practice, a UML class diagram usually fulfills the role of a fact model. One thing to check, though, is that your fact model is able to distinguish between

- Items of business record: things that are going to be stored in persistent form, probably in a database

- Objects, properties, and relationships that have definitional value but do not exist as recognizable artifacts in the operational information system or, if they do, have at most a transient existence

A convention you can use on your class diagram is to precede any names that aren't to be treated as persistent with a forward slash character (/). So, for example, a class called Entry is persistent, but a class called /EntryInput is not. Some tools may provide other ways of achieving the same end by the use of icons, colors, and so on.

Preferably, references to terms in rules should use the same form as in the fact model. For example, a complex class name might appear as MyComplexClassName. The purpose of this is to reduce the potential for ambiguity. You might want to make exceptions if the use of a precise reference of this sort obscures the business meaning of the rule, but this may equally well point to a need to find a better class name.

You'll have to decide how to refer to attributes of objects; for instance, an "order" business object may have a "total value" attribute. One style is to rely on the normal constructs available in English or whatever language you're using; for instance, "the order's total value" or "the total value of the order" or something similar. The other main option is to use dot notation, producing something like Order.totalValue. This option is much more precise and can use the names from the class diagram, but it may not be acceptable as a form of business expression. This is something that you will have to negotiate within your own organization.

You also have a choice about how to handle complex relationships. One approach is to introduce extra classes and associations into the business model to capture the necessary features. But doing this makes it more difficult to automate subsequent design steps. For instance, the class model is a good starting point for the automatic generation of a database schema, but that's going to be difficult to do if it mixes in some elements that are there only for explanatory purposes. If you do introduce additional classes and/or associations to model your constraints, you must bear in mind that these additions will almost certainly be reflected in the structure of the associated database. Changing the structure of an operational database is generally very difficult, so don't put constraints of this type into the static (class) model unless you are very sure that they are not going to change in the future.

As an example, think about the details of how many cards of various kinds could be held by a bank's customers. A fragment of a possible model is shown in Figure 3-7. Various kinds of customer-card relationships could be expressed by

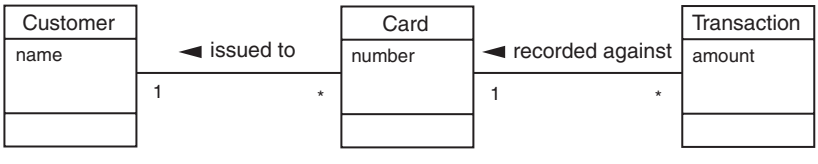


Figure 3-7 Customers, cards, and transactions

defining various specializations of customer and card. Other classes could also be introduced to reify, or make concrete, various abstract features of customers and/or cards. If treated as persistent objects, like records in tables in a database, the resulting structure could be difficult to design and implement, difficult to change, and challenging for the business owner to understand. If the relationship in the business model is kept simple, as shown in Figure 3-7, any necessary customer or card restrictions, special cases, and so on, can be expressed in a much more maintainable way as business rules.

In summary, keep the classes and relationships as simple as possible in your class diagram, and put any constraints on class membership or associations as business rules. Doing so has two advantages:

- The initial database is straightforward to design and to implement.
- Future variations in the constraints can be accommodated by changing the rules rather than the class structure, leading to much greater flexibility.

3.3.2 Individual items

The following examples navigate the relationships from the simple model in a direction that always leads to an individual item. Collections, or multiple items, are discussed later.

Classes—and implicitly the business objects they represent—are indicated by name:

```
Customer
```

If the <subject> of a rule is just an unqualified class name, the rule will apply to any object, an instance of that class:

```
A Customer must . . .
```

This could be the start of a rule that would apply to any object of type Customer.

Attributes can optionally be shown by using dot notation:

```
Classname.attribute
```

For example, we could refer to

```
The customer's name
```

or

```
Customer.name
```

whichever gives the more understandable rule.

Dot notation can be extended for more complicated relationships by using a navigation path. Starting from one class, we can navigate to another by using the

role names on the association ends. Remember that the role name defaults to the class name if not explicitly stated, although to be consistent with the UML, class names used as role names should start with a lowercase letter. In Figure 3-7, we could describe the customer associated with a particular transaction as

```
Transaction.card.customer
```

This is equivalent to the more verbose English form: "the customer to whom the card was issued that the transaction was recorded against."

Attributes can also be identified in the same way, as in:

```
Transaction.card.number
```

A feature of the dot notation is that a navigational reference to a role name has the same syntactic form as a reference to an attribute, as in:

```
Transaction.card        (a class)
Transaction.amount      (an attribute)
```

In other words, a reference of this type must be read in conjunction with a class model in order to understand what is being referred to. This feature is taken into account in the UML definitions, which forbid a role name for a class from being the same as one of its attribute names, precisely to avoid potential clashes of this sort.

Where necessary, you can qualify a reference to a class to define a rule that applies only to a subset of the class. In the following example, the <subject> of the rule is any <ContactRequest with a status of Closed>, not any Contact Request:

```
A ContactRequest with a status of Closed must . . .
```

Try to keep anything that refines the subject together in the <subject> part of the rule. Use the <fact> part of the rule for other constraints. In the following example, <System.time is greater than 16:00> does not refine the definition of a Contact Request and so is a <fact>, not part of the definition of the <subject>:

```
R312X   A ContactRequest must (something) only if System.time
        is greater than 16:00.
```

3.3.3 References to multiple items

In some cases, you will want to make statements about a collection of objects. For example, in the context of a particular customer, the following is a collection of transactions, not a single transaction (see Figure 3-7):

```
Customer.card.transaction
```

To be precise, it's all the transactions for all the cards held by the customer. Multiple levels of collections are assumed to be flattened, which is why this example is a collection of transactions, not a collection of cards, each of which has a collection of transactions.

It can feel a little awkward having a singular name when you're referring to many objects. This comes from UML's default use of the class name to designate the end of an association. This problem can be resolved by adding a specific plural role name, such as "transactions," to the association instead of using the default, singular class name. The model would then look like Figure 3-8.

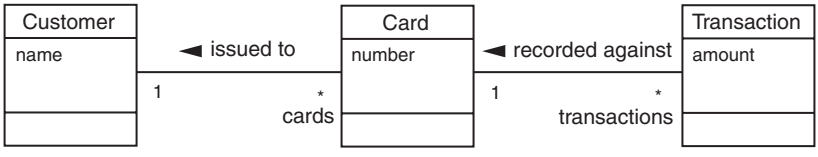


Figure 3-8 Using role names to improve readability

Several operations are relevant to a collection. Some of the most useful ones are summarized in Table 3-3. Obviously, the placeholders <collection>, <element>, and so on, can be replaced by specific references when these operations are used in a business rule. The exact form of words is less important than getting the logical statement correct. Multiple items don't introduce any concepts that we haven't already seen. In complex cases, you might want to use a technique such as Venn diagrams to get a clearer idea of the relationships between various collections and their members. The appendix gives more detail on this.

Table 3-3 Operations on collections

Operation	Usage
Size of <collection>	Defines how many elements the collection contains
Count of <element> in <collection>	Defines how many times the specified element appears in the collection
<collection> includes <element>	True if the specified element is included in the collection
<collection1> includes all <collection2>	True if all the elements in collection2 are present in collection1
<collection> is empty	True if the collection contains no elements
<element> exists in <collection>	True if at least one of the specified elements exists in the collection
<expression> for all <collection>	True if the expression is true for all the elements in the collection

3.4 Business parameters

Business rules act to constrain the operation of a system in various ways, controlling its behavior in diverse situations, ensuring that variable data always conforms to the desired structures, and so on. Business parameters are critical values that have a similar constraining influence. Rules use parameters to define boundaries, identify names used in enumerations, provide specific string values, and many other purposes. Parameters are also used by other model elements. Because they're closely related to business rules, parameters can be defined and managed in much the same way.

A parallel situation arises when common values pop up at various points in software code. If handled in the wrong way, these values seem to materialize out of thin air without visible support, so they're sometimes referred to as *magic numbers*. If the actual values are used directly in the program—called *hard-coding*—maintenance becomes more difficult. A change to a value means searching for all the places where it's been used: no mean feat if it's a large automation system. Instead, the recommended approach is to use *a program constant*. This gives a meaningful name to the value; because it is defined in just one place, it's easy to locate if a change is needed. The result in a program is something that looks like:

```
' declarations
Public Const MAX_CREDIT = 5000
     ' more code ...
If orderValue < MAX_CREDIT Then CreateOrder(items)
     ' more code ...
' and so on
```

The situation is pretty much the same at the business level. Because of the analogy with programming, these values are sometimes called *business constants*. However, this is something of a misnomer; the whole point of specially identifying these values is that they might *not* remain constant. For this reason, it's better to refer to them as *business parameters*.

Business parameters can range in complexity from a single value to a whole database table. For simple values, we can use forms of statements similar to those used for business rules, such as:

```
Maximum credit limit is $5,000.
```

For the more complicated cases, this approach soon gets very cumbersome and difficult to maintain. An example that's likely to be needed in a global e-commerce system is a set of standard parameters for individual countries. Table 3-4 shows what this might look like.

Table 3-4 Example of complex business parameters

Country	Code	Currency	Symbol	. . . and so on
Afghanistan	AF	Afghani	AFA	. . .
Albania	AL	Lek	ALL	. . .
Algeria	DZ	Algerian dinar	DZD	. . .
American Samoa	AS	US dollar	USD	. . .
. . . and so on	. . .	. . .	. . .	. . .

This information may not be as static as it seems. Countries merge, split, change their names, adopt different currencies, and so on. Any value that we might need to look up should be treated in the sense of a business parameter. The only exceptions are values that are so fixed that they don't vary from place to place or time to time; for instance, mathematical constants, such as *pi,* always have the same value anywhere in the world and for all time.

The potential for change also introduces the need for change control. If we change a parameter, such as increasing the maximum credit limit from $5,000 to $8,000, we must be able to introduce this at the right time and to align historical transactions with a particular version of the parameter. We'll look at the issues involved in configuration management in more detail in a later chapter.

Enumerations can be represented as business parameters or as attributes of special classes in the fact model as alternatives to the enumeration pattern. For example, a definition of the possible values that could be taken by a status attribute could be defined by the following rule form. (See the earlier pattern definitions for why this should be an open enumeration.)

R313 A Contact Request Status must be chosen from the following open enumeration:
—Open,
—Pending,
—Failed,
—Delivered.

Alternatively, it could be represented as a special class—the UML recommendation—as shown in Figure 3-9. You will have to decide on the most appropriate method to suit your own local circumstances.

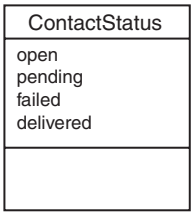


Figure 3-9 Defining an enumeration as a class

3.5 Tips on rule construction

This section discusses some common problems that arise in the construction of rule statements. The examples are intended to illustrate variations in rule syntax and so don't necessarily contain real business knowledge or relate to any particular business model. Also, each example should be taken in isolation; these rules are not intended to form a single, coherent model.

We'll look at potential problems in the following areas:

- Using facts
- Simple constraints
- Quantifications and qualifications
- States and events
- Actors
- Dangerous verbs
- Computation
- Structure and consistency

All these categories should be kept in mind when the rules are created, as well as in reviews aimed at controlling the quality of the rules, covered in more detail in Chapter 5. The example rules have been numbered for ease of reference. Rule numbers ending in X indicate rules that are incomplete, inconsistent, or badly formed.

3.5.1 Using facts

Use a fact model You should have a fact model available so that the rules can be related to other parts of the business model. The rules should always relate to elements visible in the fact model.

For example, look at the model fragment shown in Figure 3-10.

The following rule produces an ambiguity when applied to the given fact model:

R314X An Event categorized as 'Agreement' must . . .

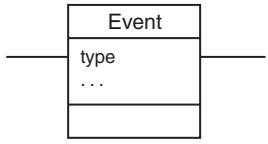


Figure 3-10 Model fragment

It's not clear whether the category stated in the rule is meant to refer to the *type* attribute of an Event business object or whether an Event is intended to have a *category* attribute as well as a *type* attribute.

The act of defining a rule may highlight some modification required to other facets of the business model, so treat the whole model as evolutionary. If it does need to be changed, don't forget to check for possible impact on other rules that have been defined earlier.

Question basic assumptions Reconsider the rule by reducing it to a simpler form by removing any associated qualifications. For example, we might have as a rule:

> R315 A loan greater than *X* must be approved by a branch manager of grade *Y* or higher with at least one year in post.

We can reduce this rule to its most fundamental form as:

> R316 A loan must be approved.

Then ask questions.

- What is the underlying business goal behind the fundamental rule?
- Does the rule fully address associated risks?
- Are there other ways of achieving the business goal?

Of course, if the rule forms part of the fundamental definition of the business, such as a "given" in the business statements of policy, there is no point in questioning the basic assumptions.

Question the terms used Look for terms that may be too wide or too narrow. For instance, in the example rule R316, we might question whether the term "loan" is too wide and whether we should be concerned only with certain types of loan, such as "unsecured loan." Alternatively, we may want to widen the term to include other financial products. As with the fundamental assumptions, if the terms are "given," there is no point in questioning them.

Explicit relationships Make relationships explicit, so that it's clear what the rule is constraining. Sentences containing the verb *to have* are a particular danger signal. You don't have to avoid *to have;* it's often the best way to state the conditions under which an attribute should be set to a particular value or business objects should be associated. Just make sure that you are being explicit.

Consider the following:

R317X Each project must have a manager.

This example does not make it sufficiently clear exactly what the project should "have" a sponsor in management? a manager representing user interests? a line manager? The rule should make this clear:

R318 Each project must be managed by a project manager.

Obscure facts or terms Look out for facts or terms that are not adequately qualified. The rule may seem clear at the time but could be open to various interpretations. This could lead to a realization that differs from your intention.

Look at this rule:

R319X All inbound contacts must identify the associated Party.

In this example, the meaning of Party might be defined adequately in the class model, but what does the word "associated" imply: the Party making the contact? the Party responsible for creating the content of the contact? the Party to whom the contact is addressed?

An improved rule might be:

R320 An inbound contact must be associated with the originating Party.

Vague terms Avoid using vague terms, such as "there," as the subject of a rule.

R321X There may be no more than two overdue payments on a type A account.

Make the meaning explicit by moving the focus of the rule to the real subject. The preceding example would be improved as:

R322 A type A account must have no more than two overdue payments.

3.5.2 Simple constraints

No permissions Avoid unqualified use of words such as "can" or "may," in a rule. Unqualified terms such as these might turn out to be permission statements

and not constraints. By definition, anything in the underlying business model "can" be true. The purpose of the rules is to define the conditions under which something in the fact model is mandated or prohibited: in other words, a constraint.

Sentences with "can" or equivalent simply say that something might or might not be the case, which is too vague to be useful. For example, the following rule is presumably trying to imply a limitation on the values of loans that can be approved by a branch manager:

R323X A branch manager can approve loans up to the value of $10,000.

But the way the rule is stated, it could be optional. It might also imply, but doesn't actually state, that the manager may not approve loans of value $10,000 and above.

The rule should state precisely what is intended, as in:

R324 A branch manager may approve a loan only if the value of the loan is less than $10,000.

Avoid padding Don't overelaborate the rules. Keep them short and to the point, and avoid padding them out with filler terms that add nothing to the definition. For example, the following rules are unnecessarily wordy ways of saying the same thing.

R325X The payment period allowed must never be greater than 60 days for any customer.

R326X The time period that's allowed for payment can be for a period of up to but not exceeding 60 days.

R327X For any order, however it has been raised, the period allowed for payment may not exceed 60 days.

A more straightforward version, containing the same information, would be:

R328 The allowed payment period must not exceed 60 days.

Using "or" Avoid joining multiple terms with "or." Such rules can be difficult to understand and to maintain. A particular trap for the unwary is the difference between *inclusive-or* and *exclusive-or;* see the appendix if you're unsure about which is which. It is usually better to split "or" rules into separate rules or to make the nature of the constraint more explicit. For example:

R329X A Party is defined as high worth if it is a VIP, has a current balance of at least *X,* or has held an account for more than *Y* years.

It's not clear whether this means exactly one or at least one of the conditions. Also, what would happen if the rule changed to two out of the three? A more explicit, and more maintainable, rule would be:

R330 A Party is defined as high worth if at least one of the following is true:
—The Party is a VIP,
—The Party has a current balance of at least *X,*
—The Party has held an account for more than *Y* years.

Of course, establishing whether a Party is to be considered a VIP may well require further rules.

Using "and" Be careful about using "and" in expressing rules. It should be reserved for cases in which you really do mean "and" in its logical sense. Maintainability is usually improved by splitting "and-ed" clauses into separate atomic rules, especially if the clauses might change independently in the future. For example:

R331X A withdrawal from an account may be made only if the account is active and the account balance is greater than zero.

This example could be expressed in more atomic form as:

R332 A withdrawal from an account may be made only if the account is active.

R333 A withdrawal from an account may be made only if the account balance is greater than zero.

If the "and-ed" terms are unlikely to change independently, another alternative would be to express the rule as a list constraint.

Complex rules Complex rules are difficult to maintain and may hide terms of business significance that should be defined more explicitly. Such rules are better split into several simple rules or perhaps even a modification of the fact model. A split may introduce a need for some new intermediate terms, each representing part of the result or a step toward its production. These new intermediate terms should be given names that have some genuine business meaning rather than placeholder names that simply link rule statements. Wherever possible, you should prefer names that are already used by the business to describe the term in question. For example:

R334X A Party with *X* successive defaults must not be offered a new loan exceeding *Y*.

This rule could be easier to maintain if the additional terms "high-risk" and "high-value" are defined in the fact model. (You would, of course, have checked that these terms are the most relevant for the business in this context.) The original single rule can then be restated more clearly in three rules:

R335 A Party with *X* successive defaults is defined as high risk.

R336 A loan exceeding *Y* is defined as high value.

R337 A high-risk Party must not be offered a high-value loan.

As well as improving clarity, this type of simplification eases maintenance of the rule population over an extended period.

Starting with "if" Avoid starting a rule with "if," which can lead to sentences that confuse the underlying logic. For example:

R338X If an account is overdrawn, the customer may make only cash transactions.

Initially, this rule sounds as though an account, not a customer, is the subject of the rule. A better version would be:

R339 A customer with an overdrawn account may make only cash transactions.

3.5.3 Quantifications and qualifications

Question qualifications Any qualifications can be assessed by stripping the rule down to its most basic form. Then, progressively add back the qualifications, while asking such questions as

- Are the qualifications necessary?
- Are they appropriate for the purpose?
- Are the specified values or ranges correct?
- Have specified values been defined as business parameters, where appropriate?

Even if the rule forms part of the fundamental definition of a business, the qualifications should still be questioned because they may have been added as embellishments during the evolution of the business descriptions.

Plural terms Wherever possible, avoid using plurals as terms of rules. For example:

R340X Contacts with high-value customers must be recorded.

Instead, express the rules as applying to individual instances of the appropriate business object. For example, the preceding rule is better as:

R341 A contact with a high-value customer must be recorded.

Each and every Don't be afraid to say *each* or *every* if it improves the clarity of the rule. Consider the following rule:

R342X An account must be allocated to a Customer Representative, with effect from 1 January 2002.

This rule might be better as:

R343 Each account must be allocated to a Customer Representative, with effect from 1 January 2002.

3.5.4 States and events

Events as subjects Avoid using a business event, such as a business action, a time, or an external trigger, as the subject of a rule. For example:

R344X By close of business, all withdrawals must be notified to the head office.

Events may well cause rules to be evaluated, but the real subject of the rule is the specific element that's affected, not the event. The preceding rule is better as:

R345 Each withdrawal must be notified to the head office before the close of business.

Ambiguous states Avoid ambiguous states. For example:

R346X The outstanding balance on a secondary account that is being closed must be transferred to the primary account before it is terminated.

Do "closed" and "terminated" mean the same thing? Does "terminated" refer to a state of the primary or to the secondary account? Each state should be distinct from other states for the same entity, and it should be clear exactly which entity the state is related to. A better statement would be:

R347 The outstanding balance on a secondary account that is being closed must be transferred to the primary account before the secondary account is closed.

Ambiguous time frames Avoid ambiguous time frames. For example:

> R348X A new account must be approved by at least two managers in the final quarter.

This rule could be taken to mean that any new account must be approved by at least two managers but not until the last quarter. The definition of the period and the nature of the constraint should both be clear-cut. The correct interpretation is more likely to be:

> R349 A new account opened in the final quarter must be approved by at least two managers.

Saying when Be wary of introducing rules that use "when," which implies that an action is fixed to a particular point. For example:

> R350X When a policy is issued, it must have a defined expiry date.

Resequencing activities and redefining their interdependencies are typical areas for business process reengineering. This makes the "when" better defined in a business event or a business process, not in a business rule. A rule statement should refer to a constraint that applies in a particular state, not to transitions between states.

It might be possible for the expiry date to be changed after a policy has been issued. We want to eliminate the possibility that a subsequent change may result in a null expiry date, so a better version would be:

> R351 An issued policy must have an expiry date.

3.5.5 Actors

Question actors If actors appear in the rule, we should ask questions about why that particular actor needs to be specified. For instance, in rule R324, we might question why the approver should specifically be a branch manager. Again, it may be appropriate to consider a narrower or a wider definition.

Consideration of this point may reveal a need for a better classification of actors. This is important because business process engineering activities often involve reassignment of tasks from one actor or group to another.

Even if the rule forms part of the fundamental definition of a business, it's still worth questioning the actors. For example, they may just reflect the current status quo rather than being genuine requirements of the business.

Actors as subjects If you can, avoid making actors the subjects of rules. As we've said, redefining which actor does which action is a common feature in business process reengineering.

R352X A customer representative may issue a replacement charge card only if the old card expired within the last 30 days.

Putting this definition of who does what into the process descriptions makes it easier to change workflow patterns. Leave the rules to express the actor-independent logic. Thus, this rule would be better as:

R353 A replacement charge card may be issued only if the old card expired within the last 30 days.

3.5.6 Dangerous verbs

Command verbs Avoid command verb forms. These often appeal to an invisible actor that may or may not exist. Such forms may also indicate related activity that's probably better captured in a process description. For example:

R354X Don't credit an account with a deposit amount if the owning Party is in default.

This rule would be better as:

R355 An account for a Party in default may not be credited with a deposit amount.

Action verbs Avoid action verbs, which are likely to create unclear definitions. For example:

R356X A facility must not be terminated if transactions are pending.

Remember that rules should constrain what must or must not be true in a particular state; they are not intended to be descriptions of processes. Thus, this rule is better as:

R357 A terminated facility must not contain pending transactions.

CRUD words Look carefully at rules that use CRUD words (create, read, update, delete) or other terms relating to possible implementations. Again, these words can refer to an action rather than to a constraint and may represent a procedure that should be scheduled. For example:

R358X Update the account current balance by adding the new deposit amount when received.

Rules should be eventless and express only the underlying business logic. Thus, this rule would be better as:

R359 The account current balance is defined as the previous balance plus the amount of the new deposit.

CRUD-type words may provide useful clues in rule discovery, but the rule statements should decouple the constraint from its point of application.

3.5.7 Computation

Ambiguous computation You can clarify the essence of a computation by making the result of the computation the subject of the rule. For example:

R360X A customer may hold no more than three Type A and Type B accounts combined.

In this example, the computation implied is not immediately clear but is more obvious if the total is made the subject of the rule, as in:

R361 The combined total of Type A accounts and type B accounts held by a customer must not exceed three.

Embedded computations Look out for computations embedded within rules; separate these computations out. An algorithm or a formula might need to change independently from the entity that's being constrained by the rule. For example:

R362X The sum of the repayments to date must be greater than or equal to the cumulative debt.

Separating these concerns makes the rules easier to maintain. This rule could be easier to maintain if expressed as two rules:

R363 The cumulative repayment value for an account is defined as the sum of all repayment amounts to date.

R364 The cumulative repayment value must be greater than or equal to the cumulative debt.

3.5.8 Structure and consistency

Missing rules You can identify missing rules by thinking carefully about the kinds of constraints that would make business sense. A good way to do this is to work systematically through your business objects, especially the relationships among them. At each point, you can ask such questions as, "Does this always

apply?" "Are there any special cases or exceptions?" "Is there perhaps a particular situation in which unrelated objects X and Y *could* be related?"

Overlap Look among rules that relate to the same part of the model to check for overlapping coverage, where one rule is wholly or partly contained within another. In such cases, it may be necessary to rationalize the rules to remove the overlap. Take, for example, the following two rules:

R365 A loan exceeding $1,000 must be approved by a branch manager or above.

R366X A loan exceeding $1,500 must be approved by a branch manager or above.

We can see that the second rule is subsumed by the first, so we could eliminate the second rule without losing any of the business control that the rule implies. Of course, we would also need to look at the source of both rules to make sure that $1,000 is indeed the right value to use in the single rule.

Note that this doesn't contravene the guideline about rules not specifying who applies them. Rule R365 expresses a relationship between a range of loan values and the seniority of the approver. The rule could be applied by a customer service agent checking a set of forms, an automated information system, an external auditor, or perhaps all these.

Duplication Look for rules that are the same or very similar, perhaps with slight differences in the terms used or a different ordering of the statement wording. For example, the following two rules are duplicates and should be replaced by one rule.

R367X The total value of all current loan facilities must not exceed *X*.

R368X The sum of the values of existing loan facilities must not be greater than *X*.

This situation can arise if the current repository is not properly checked before defining a new rule or where an existing rule has been poorly categorized, making it difficult to find. Use of the standard rule patterns and rooting the rules in a fact model should minimize the occurrence of duplication.

Inversion Look for situations in which two rules produce the same result, but one rule is stated as the inverse or complement of the other. For example:

R369X A new loan may not be offered to an overdrawn customer.

R370X An overdrawn customer may not be offered a new loan.

These two rules represent the same logic but with the order of terms reversed.

Inversions are particularly likely to appear in the following situations:

- If A exists, then B must exist; If B exists, then A must exist.
- A must be equal to B; B must be equal to A.
- A must be greater than (or less than) B; B must be less than (or greater than) A.

The same is true for the negative (must not) forms.

Take special care with "greater than" and "less than" statements to consider whether the "equal to" case is meant to be included or excluded. Also, check carefully what is really intended by the rule statements. The appendix discusses how logical statements can legitimately be turned into alternative forms and some fallacies arising from incorrect transformations.

Conflict Conflict arises when two or more rules produce contradictory results. For example:

R371X An account holder may be issued with only one card.

R372X An account holder may be issued with no more than two cards.

These two rules would produce conflicting results if an attempt were made to issue a second card to an account holder. Conflicts of this sort must always be resolved; rules with the potential for conflict cannot coexist within the same population.

Rule references All the references between the rule statements and the rest of the model should be complete and consistent. "Complete" is difficult to be sure about. It would be nice to have the time to gently ruminate on the relationship of everything to everything else, but the realities of life are otherwise. In practice, you're limited to staying alert during the specification stages, especially at review meetings. Some tool support to show what's related to what can help, but it won't do the job for you. "Consistent" is a property that can be helped by some proactive measures, such as the enforcement of naming standards and thorough reviews—again, with decent tool support.

All this applies both to references made from the rules to other model elements, such as to business objects, and to references made to the rules from other elements, such as from use case descriptions. If you have any particular conventions about naming or numbering, they too must be observed.

Repository information All the relevant fields in the rules repository should be checked to make sure that they have been completed appropriately with the correct information. In addition to the rule definitions themselves, this also covers such fields as supporting notes to provide documentation of complex or unusual cases.

Here, too, good tool support is invaluable. Even something as simple as identifying all the places that a particular term has been used can make a crucial difference to your ability to impose control over a swelling rule population. We'll return to this topic again in later chapters.

3.6 Case Study: Microsoft Outlook

Microsoft Outlook is a desktop program that's designed to help you to manage messages, appointments, contacts, and other facets of modern business life. The program integrates with other Microsoft applications, such as Exchange, to provide a comprehensive range of facilities. Here, we're going to look at the way that Outlook uses rules to automate the handling of mail messages.

Rules in Outlook are composed by using a "wizard" that provides an interactive dialog from which you can define the various parts of your rule. The interface design shows how a potentially complex and technical task can be made much easier for ordinary users, but we're going to concentrate on the way that the rules are structured and the logic that lies beneath them.

Taking Outlook as an example isn't meant to imply that it's a pure expression of an idealistic rule concept. In fact, it violates some of the guidelines that we've already been over, because Outlook, like the rest of us, has to accommodate the realities of today's information systems. Sometimes, the ideal has to be sacrificed for the workmanlike. Among other things, this case study shows the sort of tradeoffs you may need to make between the pure and the pragmatic. From that point of view, Outlook is a pretty good example of the kind of compromise that you may need to find in your own applications.

3.6.1 Outlook rule structure

Each rule statement is made up of a number of clauses. The initial type clause defines whether the rule applies to outgoing or to incoming mail messages. There can be zero or more condition clauses, selected from a range of possibilities. When specified, all these must be true for the rule to be activated. At least one action clause must define the action to be taken if the rule is activated. Finally, zero or more exception clauses can be selected from a range of possibilities. When specified, all the exception clauses must be false, or the rule will not be activated. In summary, a rule statement has the following general structure.

```
typeClause    [conditionClause ["and" conditionClause]]
              actionClause ["and" actionClause]
              ["except" exceptionClause ["or" exceptionClause]]
```

A typical rule statement constructed in this way might be:

```
Apply this rule after the message arrives
where my name is in the To or Cc box
        and which has an attachment
delete it
except if sent only to me
        or except if it is an Out of Office message
```

All the possible conditions, exceptions, and actions are drawn from predefined lists. The condition and exception lists are logically the same, but the text strings are slightly different to create a more natural reading of the rule.

The reason for having both conditions and exceptions is that it's sometimes more convenient to use one or the other. A rule with a large number of conditions might be expressed equivalently by a rule with a small number of exceptions and vice versa. For instance, you could say something like:

> "apply this rule if <condition> and <condition> and . . . and <condition>. . . ."

But you would probably find it more natural to say something like:

> "apply this rule except where <exception>. . . ."

Outlook allows you to define rules by using any combination of conditions and exceptions, but it won't necessarily identify problems if you chose an inconsistent combination. For example, what would happen if you use the same clause as both a condition and an exception? It's unlikely that you would make such a mistake, because your Outlook rules will probably be simple enough to make such an error stand out. However, it may be less obvious in more complex environments. Remember that the underlying logic may still produce a result in a case like this: It just might not be the one that you expected.

3.6.2 Conditions, exceptions, and actions

Table 3-5 lists the conditions and exceptions available in the version of Outlook shipping with Microsoft Office 2000. The actions that can be taken, and therefore the action clauses, are also dependent on the type of mail message—incoming or outgoing—and these are summarized in Table 3-6.

Table 3-5 Outlook conditions and exceptions

Condition or Exception Clause (exception wording is slightly different)	Type Clause: Apply this rule after the message arrives	Type Clause: Apply this rule after I send the message
sent directly to me	Y	
sent only to me	Y	
where my name is in the Cc box	Y	
where my name is in the To or the Cc box	Y	
where my name is not in the To box	Y	
from people or distribution list	Y	
sent to people or distribution list	Y	Y
with specific words in the recipient's address	Y	Y
with specific words in the sender's address	Y	
with specific words in the subject	Y	Y
with specific words in the body	Y	Y
with specific words in the subject or body	Y	Y
with specific words in the message header	Y	
flagged for action	Y	
marked as importance	Y	Y
marked as sensitivity	Y	Y
assigned to category	Y	Y
which is an Out of Office message	Y	
which has an attachment	Y	Y
with selected properties of documents or forms	Y	Y
with a size in a specific range	Y	Y
received in a specific date span	Y	
uses the form name form	Y	Y
suspected to be junk e-mail or from Junk Senders	Y*	
containing adult content or from Adult Content Senders	Y*	

*Can be used in conditions only, not exceptions

Table 3-6 Outlook actions

Action Clause	Type Clause: Apply this rule after the message arrives	Type Clause: Apply this rule after I send the message
move it to the specified folder	Y	
move a copy to the specified folder	Y	Y
delete it	Y	
forward it to people or distribution list	Y	
reply, using a specific template	Y	
notify me, using a specific message	Y	
flag message for action in a number of days	Y	Y
clear the Message Flag	Y	
assign it to the category category	Y	Y
play a sound	Y	
mark it as importance	Y	Y
mark it as sensitivity		Y
notify me when it is read		Y
notify me when it is delivered		Y
cc the message to people or distribution list		Y
defer delivery by a number of minutes		Y
perform a custom action	Y	Y
stop processing more rules	Y	Y

In some cases, the clauses include references to parameters that refine the meaning of the clause. Initially, a placeholder marks the position of the parameter, shown underlined in the tables. The placeholder is a temporary string that makes sense within the rule statement but has not yet been made specific. Later, this placeholder string is replaced by a string that defines the actual value or reference. As with the placeholder, the new embedded string is worded to make for a natural reading of the rule statement. Figures 3-11 and 3-12 show an example of a rule before and after the placeholder values have been refined. The placeholders Outlook uses in condition or exception clauses—indicated in the *c/e* column—or in action clauses—indicated in the *a* column—are summarized in Table 3-7.

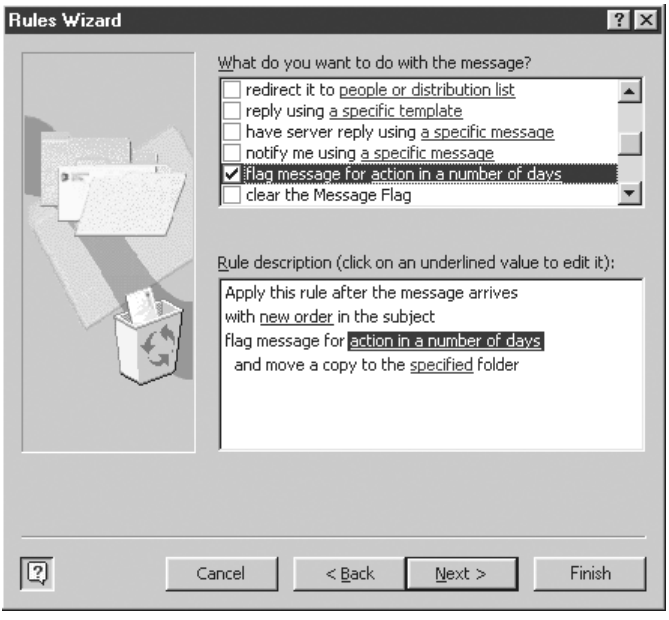


Figure 3-11 Outlook rule with placeholder

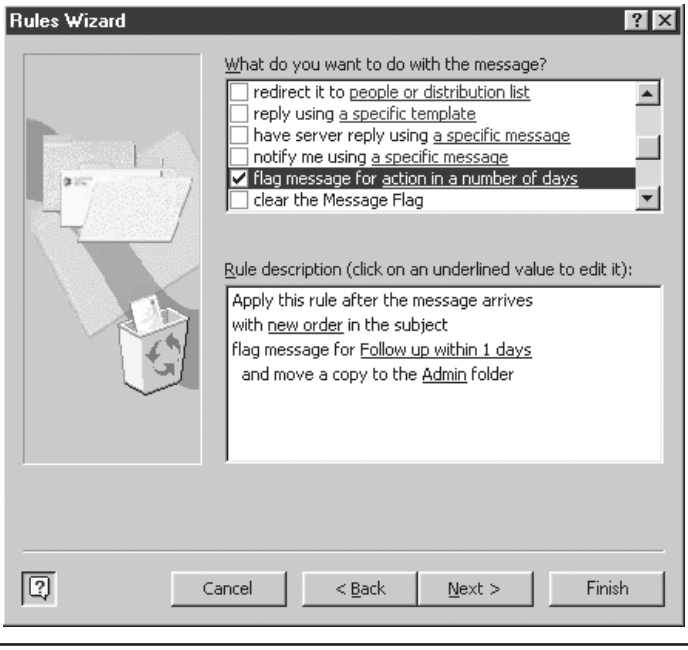


Figure 3-12 Placeholder replaced by value

Table 3-7 Outlook placeholders

Placeholder	Usage	c/e	a
people or distribution list	Entry from normal Outlook mail addresses	Y	Y
specific words	Text string	Y	
action	Entry from list of Outlook message flag types	Y	
importance	Selected from Outlook importance types: Low, Normal, High	Y	Y
sensitivity	Selected from Outlook sensitivity types: Normal, Personal, Private, Confidential	Y	Y
category	Names of categories, separated by semicolons	Y	Y
selected properties	List of properties of documents of forms, with the test to be applied to each property (equal-to, less-than, and so on)	Y	
specific range	Maximum and minimum file size, in kilobytes	Y	
specific date span	After date and/or before date	Y	
form name	Form selected from available form names	Y	
Junk Senders	Entry from list of known junk e-mail senders	Y	
Adult Content Senders	Entry from list of known junk e-mail senders (same list as Junk Senders)	Y	
specified folder	Entry from Outlook's Folder List		Y
specific template	Entry from list of available Outlook templates		Y
a specific message	Text of message		Y
action in a number of days	Message flag type and number of days		Y
a sound	Name of sound file		Y
a number of	Number of minutes		Y
a custom action	Predefined custom actions		Y

3.6.3 Internals

Internally, Outlook contains code to evaluate the rule statements and to carry out the actions. Because this code has to do only one job, it can be optimized for its given role. Without knowing the details of the implementation, you can guess at the likely internal organization by thinking about how you might go about designing the rule subsystem.

The internal representation of rule clauses need not be exactly the same as displayed on the screen, providing that the logical sense is preserved. You can make structures of this sort more efficient by using tokens to represent the various types of clauses (condition, exception, action) instead of the literal text string that was selected by the user. Each token would be linked internally to the following:

- A text string, for display to the user
- Zero or more links to additional parameters, to refine the meaning of the clause

There's also an implied link to some program code that's able to establish true/false values for conditions and exceptions or to carry out the associated action. You could realize this by having your program branch to a particular point in the code when it encounters a particular token type, but we won't bother with the details here.

For a complete rule, you would need to store

- Its name (a text string)
- A token to indicate the type (incoming or outgoing mail message)
- Zero or more condition tokens
- Zero or more exception tokens
- One or more action tokens

You might also choose to store some additional information with each rule. Management information, such as the date and time last edited and the name of the user, can be useful for maintaining the rules. Auxiliary information, such as cached intermediate results, can make the implementation more efficient. However, these don't affect the logical structure, so, again, we'll skip over the details.

3.6.4 Logic

In a similar way, the logic structure need not be evaluated exactly as laid out on the screen. In a closed environment like this, we know that rules are always going to have a consistent structure, so we can always process them in a consistent way. We can write down a simple formula that defines how the clauses have to be processed, using basic logic constructs: `And`, `Or`, and `Not`. Turning the formula into program code is then a straightforward job.

The basic logic structure of a rule in Outlook can be summarized in codelike form as:

```
If      ( And (c1, c2, c3, ... , Not ( Or (e1, e2, e3, ... ) ) ) )
Then    ( a1, a2, a3, ... )
```

where:

- $c_1, c_2, c_3, \ldots$ are conditions; all must be true for the rule to apply.
- $e_1, e_2, e_3, \ldots$ are exceptions; all must be false for the rule to apply.
- $a_1, a_2, a_3, \ldots$ are actions that must be carried out if the rule applies.

You'll see that this is slightly different from the user's version. On screen, the actions are shown between the conditions and the exceptions of the rule, so that the statement reads more naturally to a human user. However, when you're considering mechanization of the logic, it makes sense to put the statement into a more mathematical form.

Now that we have the stark outline of the logical structure of a rule, it's easy to answer the question we posed earlier. What would happen if we used the same clause as both a condition and an exception? The test applied to decide whether to carry out the action(s) would look something like:

```
And (A, B, C, ... Not ( Or (A, P, Q, ...) ) )
```

where `A`, `B`, `C`, and so on, stand for the clauses, with `A` being the clause that's common. If we assume that `A` is true, the `Or` part must also be true. Because Not (true) is false, we're looking at the truth value of the whole expression as being something like:

```
And (True, ... False)
```

which will always be false. Similarly, if `A` is false, we're looking at a truth value like:

```
And (False, ... ?)
```

The query indicates that the value of the `Or` part depends on the other exception clauses, but this has no effect on the overall result; it will always be false. In other words, whatever combination of values we have for the conditions and exceptions in this rule, the associated actions will *never* be carried out.

This illustrates the value of having a clear structure to the underlying logic. We did not need to consider all the possible condition and exception clauses, and maybe their parameters too, in order to come to this conclusion. Just knowing the logic formula—and that a clause must be either true or false—we could establish how Outlook will behave in every instance of this situation.

3.6.5 Outlook rule features

Now that we've seen something of how Outlook uses messaging rules, let's consider how it compares with the general principles we outlined earlier. Here are some of the features of the Outlook rules that you should have been able to spot from the description.

- Rules are built from a range of predefined patterns. The patterns are chosen to be relevant to the nature of the domain; in other words, they allow Outlook users to say what they need to say.
- Rule statements are expressed in constrained natural language. Because the rules can be read as simple sentences, they are easy for nonexperts to create and to understand.
- The rule clauses use canned text to avoid the problems of natural-language processing. Users have no option to create free-format text, and so the clauses available can be designed in advance to be clear and meaningful in the places they appear.
- Rules may make references to terms that have a standard meaning in the context of the domain. The context is electronic mail, so the terms cover such objects as *sender* and *message* with such properties as *address* and *subject,* which make sense in that domain.
- All defined rules are maintained in a repository. Rules are identified by descriptive names and can be checked out for editing, if required. Some meta-information, such as whether a rule is active, can also be viewed and edited.
- The internal representational structure does not necessarily line up one to one with the rule statement structure. Once a rule or a set of rules has been defined, the system can exploit knowledge of the rule structure to turn the statements into an appropriate form for processing.
- Internally, rule clauses are combined strictly in accordance with standard logic. There's no need for special-to-purpose algorithms. Evaluating the logic expressed by the rule statements is all that's needed.

All this is pretty much in accord with the ideas discussed earlier in this chapter. There are some differences, though, arising from the size and the scope of the application.

- There's no explicit fact model. Because the scope of the application is limited, it's not too difficult to remember what all the terms mean, and it's very unlikely that users would disagree about them. This would not scale up to a full business system, for which a fact model would be necessary to provide an agreed point of reference.
- There's no opportunity to use rules for any other purpose or in any manner but the way they're hard-wired into Outlook. Here the rules are contained inside the application and invoked at a fixed point in its operation. This is a

trade-off worth making; the resulting gain in simplicity far outweighs the uncertain benefits that might be gained from using Outlook features elsewhere.

- The conclusion of the rule is tied immediately to a procedural action. This departs from the declarative style recommended as a general practice, but, again, the additional complexity would be difficult to justify in the context of the Outlook application.

Although it is not a complete business system, Outlook provides a useful example of a small-scale usage of business rules. As we'll see in later chapters, using rules on a wider scale requires some industrial-strength features that are not necessary in Outlook, particularly when dealing with the practicalities of using rules in distributed information systems.

3.7 Rule description summary

Rules are used to describe the conditions that *must* apply when the system is in a stable state. In other words, a rule violation may exist temporarily, but the system should take some automatic corrective action whenever such a situation is detected.

For business rules to be used effectively, a wide range of people must be able to read and to write them without needing special technical expertise. The best way of achieving this is for the original source rule statement to be expressed in a fairly colloquial natural-language form. Many of the difficulties of unconstrained natural-language processing can be avoided by defining a suitable set of rule patterns to be used as templates.

Behind the scenes, the source rules can have other, equivalent, forms that might be better suited to subsequent automation activities. Regardless of how this is achieved, it must still be possible to create and to maintain the rules in their more natural form.

The terms referred to in the rules must have a consistent and well-defined interpretation. This can be achieved by insisting on a supporting fact model in which the business rules can be rooted.

Rule statements are a key element in defining the intentions and the needs of the business. Many downstream decisions will depend on what the rules say, so it's worth spending some up-front effort on making sure that they are accurately stated and properly aligned to the aims of the business.

Index